Alocare dinamică

Capitolul

9

- **Generalități**
- **Alocare statică şi alocare dinamică**
- **Structuri semistatice**
- **Structuri dinamice**
- **Implementarea structurilor de date folosind alocare dinamică**
- **Structuri arborescente**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

9.1. Generalități

Tipurile de date de care avem nevoie pentru a putea memora şi prelucra datele, fie există în limbajul de programare ales, fie se pot declara cu ajutorul tipurilor existente. În organizarea datelor, datele de diferite tipuri se grupează în *structuri de date.* Dacă gruparea se face la nivelul rezolvării problemei, independent de mediul de stocare a datelor, vorbim de *structuri abstracte (logice) de date*.

Problema stocării acestor structuri ne conduce la existența unei *reprezentări fizice* a structurilor logice de date. Având în vedere că datele trebuie să fie disponibile în memoria internă în timpul executării programului şi că această memorie se consideră organizată liniar, se pune *problema liniarizării structurilor de date*. Această liniarizare se realizează în mod diferit pentru structuri de date diferite, astfel încât operațiile elementare care se efectuează asupra lor să se poată efectua în timp optim.

O structură de date poate ocupa în memorie o zonă de dimensiune constantă, în care elementele componente ocupă tot timpul executării programului acelaşi loc. O astfel de structură este *statică*.

Dacă o structură de date ocupă o zonă de dimensiune constantă, dar elementele componente ocupă un loc variabil în timpul executării programului, atunci o astfel de structură este *semistatică*.

Dacă o structură de date ocupă în memoria internă o zonă care se alocă în timpul executării programului pe măsura nevoii de prelucrare, fără a avea o dimensiune constantă, atunci structura este *dinamică*.

Alocarea de memorie pentru diferite tipuri de structuri de date se realizează în mod diferit de către compilatoare.

- În cazul structurilor statice alocarea este *statică atât la nivelul întregii structuri, cât şi pentru fiecare componentă în parte*.
- Pentru o structură semistatică alocarea este *statică la nivelul structurii şi dinamică la nivelul componentelor.* Acestea se pot aloca fie static, fie dinamic.
- Pentru o structură dinamică alocarea este *dinamică la nivelul componentelor cât şi la nivelul întregii structuri*.

Observaţie

Această clasificare a structurilor de date în structuri statice, semistatice şi dinamice se referă la *structurile fizice* de date şi nicidecum la însăşi aceste structuri. *O aceeaşi structură abstractă de date poate fi, în general, implementată şi ca structură statică şi ca structură dinamică.*

Există structuri logice de date ale căror implementare în mod consacrat este fie static, fie dinamic. De exemplu, o mulţime se va implementa, de regulă static. O stivă poate fi implementată static sau dinamic, asigurând o implementare adecvată a operaţiilor specifice de adăugare şi eliminare a elementelor. Este important să se găsească acea implementare care convine cel mai bine aplicaţiei considerate.

9.2. Alocare statică şi alocare dinamică

În studiul realizat până acum am întâlnit doar variabile alocate static. Acestora li se alocă spaţiu în timpul compilării, spaţiul respectiv rămâne „ocupat" până la sfârşitul execuţiei programului respectiv. Am pus ghilimele, deoarece nu întotdeauna vom ocupa efectiv cu date de prelucrat o astfel de zonă de memorie alocată. Rezervarea spaţiului se realizează în timpul compilării, pe baza declaraţiilor, urmând ca în timpul execuţiei programului să se lucreze cu date care ocupă întreaga zonă rezervată sau numai o parte din aceasta. De aici rezultă imediat prima deficienţă a acestui tip de alocare, deoarece este evident că în timpul prelucrării acest spaţiu poate să se dovedească fie prea mare, fie prea mic.

Cealaltă inconvenienţă se referă la faptul că dimensiunea spaţiului alocat rămâne nemodificat pe tot parcursul execuţiei programului. Altfel spus, dacă într-un program prelucrăm o structură de date care creşte pe parcursul execuţiei, eventual şi o structură care descreşte, pentru ambele trebuie să alocăm spaţiu la dimensiunea maximă posibilă. Dacă într-o astfel de structură (de exemplu într-un tablou unidimensional) trebuie să inserăm elemente noi, la începutul ei sau undeva între două elemente, vor fi necesare numeroase translatări. Invers, dacă din aceste structuri vrem să eliminăm elemente, din nou vom efectua translatări care conduc la mărirea timpului de execuţie.

Aceste probleme, practic, pot să apară în cazul prelucrării acelor structuri de date ale căror dimensiune se modifică în timpul prelucrărilor. Ele, de regulă, se alocă dinamic. Alocarea dinamică este o alternativă de a lucra cu resursele memoriei care oferă

posibilitatea ca programatorul să se „extindă” cu datele de prelucrat într-o zonă specială de memorie, numită *heap*. Aici se pot aloca variabile simple, tablouri, dar structurile de date specifice sunt cele numite în introducerea acestui capitol semistatice şi dinamice. Reamintim că orice structură de date poate fi alocată fie static, fie dinamic. Rămâne la latitudinea programatorului, ca în funcţie de următoarele criterii să aleagă modalitatea de alocare. De regulă, se lucrează cu alocare dinamică, dacă:

- *nu se cunoaşte dimensiunea* structurii de date în momentul începerii execuţiei programului;
- dimensiunea structurii de date *variază* în timpul execuţiei;
- se vor efectua operaţii de *inserare* respectiv *eliminare* de elemente în/din structură;
- structurile de date *nu au suficient spaţiu în memoria statică*, unele trebuie plasate în *heap*.

9.3. Structuri semistatice

Datele de tip tablou, mulţime şi articol (înregistrare), întâlnite până acum sunt structuri de date statice care, de regulă, se prelucrează alocând spaţiul necesar în mod static. Dar am întâlnit în clasa a 9-a şi o structură semistatică, *stiva*.

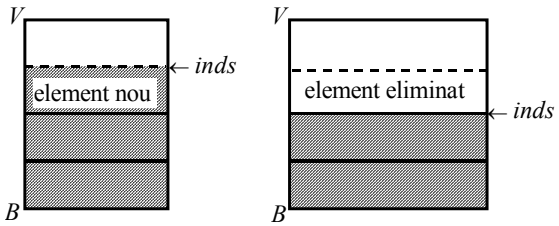
Pe parcursul rezolvării problemelor care prelucrau stive (în clasa a 9-a) am utilizat alocare statică, declarând stiva sub forma unui tablou unidimensional. Am realizat operaţii de adăugare şi de eliminare elemente, actualizând dimensiunea tabloului după fiecare astfel de operaţie. Atunci când un element nu a mai fost necesar, el a fost „eliminat” din structură, iar locaţiile corespunzătoare au devenit libere (puteau fi reutilizate).

Când structura semistatică nu conţine nici un element, ea este *vidă*, iar când întreg spaţiul alocat a fost utilizat, ea este *plină*. În cazul alocării statice, înainte de o operaţie de adăugare a unui element nou am verificat dacă structura nu este plină, iar înainte de o operaţie de ştergere, am verificat dacă structura nu este vidă.

În acest capitol vom aborda implementări dinamice şi pentru structurile semistatice, exemplificând diferitele operaţii în limbajul Pascal.

9.3.1. Stiva

Reamintim că stiva (din punct de vedere logic) se alocă într-un spaţiu S având baza B şi vârful V, astfel încât $B < V$.



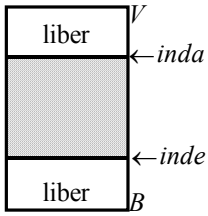
a) Adăugarea unui nou element se va face prin alocarea primei locaţii libere de la bază către vârf;
b) Ştergerea unui element se face prin eliberarea primei locaţii ocupate de la vârf către bază.

Aceste operaţii necesită păstrarea şi gestionarea unui *indicator de stivă* care creşte la adăugare (*inds* ← *inds* + 1) şi descreşte la ştergere (*inds* ← *inds* – 1). Aşadar, stiva poate fi privită ca o structură semistatică formată din cuplul (*S*, *inds*).

Dacă stiva este *vidă*, *inds* = *B*, iar dacă stiva este *plină*, *inds* = *V*. Înainte de adăugare, respectiv eliminare, se va verifica dacă stiva nu este plină, respectiv vidă. Modul în care intră şi ies elementele din stivă justifică şi denumirea de *listă LIFO* (***L**ast **I**n – **F**irst **O**ut*).

9.3.2. Coada

Cealaltă structură semistatică, frecvent utilizată în prelucrarea datelor este coada. Fie *S* o zonă de memorie având baza *B* şi vârful *V*, astfel încât *B* < *V*. Avem *V* – *B* = *n*. Coada poate fi privită ca o structură semistatică formată din tripletul (*S, inda*, *inde*), unde *inda* şi *inde* sunt doi indicatori relativi, care arată locul unde trebuie adăugate elemente noi, respectiv de unde se elimină elementele. Adresa elementului de extras şi adresa la care se pune elementul de adăugat va fi *B* + *inde* respectiv *B* + *inda*.



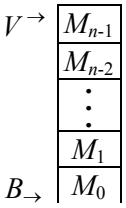
Coada este plină dacă *inde* = (*inda* + 1)(mod *n*) şi este vidă dacă *inda* = *inde*. Iniţial *inda* = *inde* = 0 (coada vidă). La adăugare *inda* ← (*inda* + 1)(mod *n*), iar la eliminare *inde* ← (*inde* + 1)(mod *n*). Înainte de adăugare se verifică dacă structura este plină, iar înainte de ştergere se verifică dacă avem sau nu coadă vidă.

Datorită modului în care se adaugă şi se extrag elementele, coada se mai numeşte şi *listă FIFO* (***F**irst **I**n – **F**irst **O**ut*).

9.3.3. Tabela de dispersie

Amintim şi tabela de dispersie care, de asemenea se alocă într-un spaţiu *S* având baza *B* şi vârful *V*, astfel încât *B* < *V*, împărţit în *n* părţi egale, numerotate cu valori aparţinând mulţimii {0, 1, ..., *n* – 1}. Fie *M* o mulţime şi o funcţie *h*: *M* → {0, 1, ..., *n* – 1} definită pe *M*.

Tabela de dispersie T este o structură semistatică prin care se realizează partiționarea mulțimii M în n clase disjuncte: M_0, M_1, ..., M_{n-1}.



Cel mai simplu exemplu de tabelă de dispersie este acela în care elementul $m \in M$ se caută mai întâi la adresa $B + h(m)$, apoi la $B + (h(m) + 1)(\text{mod } n)$, apoi la adresa $B + (h(m) + 2)(\text{mod } n)$ ş.a.m.d. Datele din părțile tabelei, în general, sunt memorate sub forma unor liste liniare. Căutarea se termină în unul din următoarele trei cazuri: elementul căutat a fost găsit, celula $B + (h(m) + i)(\text{mod } n)$ este liberă sau au fost parcurse toate cele n locații.

9.4. Structuri dinamice

În cazul structurilor dinamice alocarea dinamică a spațiului de memorie necesar se realizează pe măsura apariției elementelor structurii. Cu toate că există posibilitatea alocării în heap şi a unor variabile simple sau a unor tablouri, alocarea dinamică, de regulă, impune *înlănțuirea* acestora. Reprezentarea internă a unui element trebuie să conțină, pe lângă *informația utilă propriu-zisă,* şi o informație suplimentară – numită *referință* – prin care se realizează înlănțuirea.

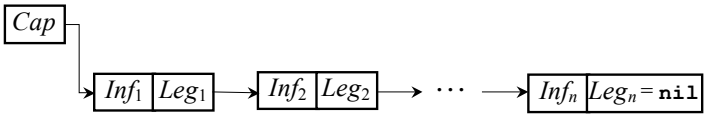
Informație utilă (propriu-zisă)	Informație de legătură (referință)

Dacă într-o structură dinamică toate elementele sunt pe un singur nivel, ea se numeşte *listă liniară*, dacă elementele sunt structurate pe mai multe niveluri, avem o *structură arborescentă*, iar dacă elementele nu sunt grupate pe niveluri avem *structuri de tip rețea*.

9.4.1. Lista liniară

Fiind dată o mulțime M, prin *listă liniară* se înțelege o secvență (eventual vidă) de elemente din M. Pentru reprezentarea internă a unei liste vor fi necesare:

a) *referință la primul element* din listă (numit *cap de listă*);
b) *înlănțuirea* elementelor listei între ele;
c) *indicarea ultimului element* din listă (prin referința specială `nil` care semnifică faptul că în înlănțuirea respectivă, după acest element nu mai urmează alt element).



Listele de acest tip sunt liste liniare *simplu înlănțuite*. Dacă informația de legătură conține atât adresa elementului *precedent* cât şi adresa elementului *următor*, avem liste *dublu înlănțuite*. Dacă într-o listă liniară primul şi ultimul element sunt legate astfel încât Leg_n = *referință la primul element*, lista se numeşte *circulară*.

Listelor liniare alocate dinamic le sunt caracteristice operațiile: inserare şi eliminare element în/din structură, crearea structurii prin inserare în lista inițial vidă, traversarea listei, căutarea unui element, concatenarea sau tăierea în două (eventual în mai multe părți) a unei liste.

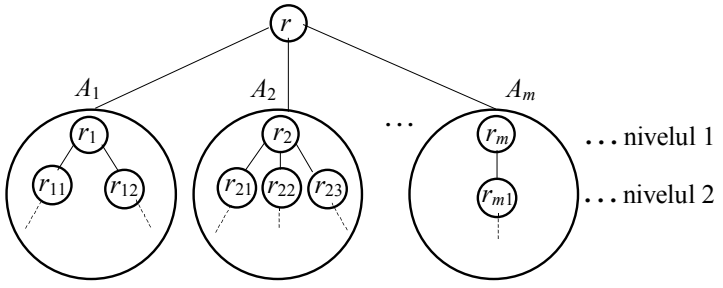
9.4.2. Arborele

Fiind dată o mulțime M de elemente denumite *noduri* sau *vârfuri*, vom numi *arbore* (conform definiției date de *Knuth*) un set finit de noduri astfel încât:

a) există un nod cu destinație specială, numit *rădăcina* arborelui;
b) celelalte noduri sunt repartizate în $m \geq 0$ seturi disjuncte $A_1, A_2, \ldots A_m$, fiecare set A_i constituind la rândul său un arbore.

Elementele arborelui sunt *nodurile* şi *legăturile* dintre ele. Nodul rădăcină r îl considerăm ca fiind pe *nivelul* 0.

Întrucât $A_1, A_2, \ldots, A_m$ sunt la rândul lor arbori, fie $r_1, r_2, \ldots, r_m$ rădăcinile lor. Atunci $r_1, r_2, \ldots, r_m$ formează nivelul 1 al arborelui. Nodul r va avea câte o legătură cu fiecare dintre nodurile $r_1, r_2, \ldots, r_m$. Continuând acest procedeu, vom avea nivelurile 2, 3, ..., ale arborelui. Dacă numărul nivelurilor este finit, atunci şi arborele este *finit*, în caz contrar se numeşte *arbore infinit*.



Această viziune asupra arborilor este ierarhică, deoarece nodurile sunt subordonate unele altora. Fiecare nod este subordonat direct unui singur nod, excepție constituie rădăcina care nu este subordonată nici unui nod. Dacă un nod nu are nici un nod subordonat, atunci se numeşte *frunză* sau *nod terminal*.

O categorie foarte răspândită de arbori o reprezintă *arborii binari*. Un arbore binar se compune din rădăcină şi din doi subarbori denumiţi *subarborele stâng*, *respectiv drept*. Amintim faptul că există o deosebire între *arborii binari* şi *arborii oarecare având noduri cu cel mult doi descendenţi*. În cazul arborilor oarecare este suficient dacă se ştie că *B* este fiul lui *A*, pe când la un arbore binar trebuie să ştim în plus dacă *B* este fiul stâng sau fiul drept al lui *A*.

Dacă nodul *A* are ca subordonaţi nodurile *B* şi *C*, atunci *A* se numeşte părintele (tatăl) lui *B* şi *C*, *B* şi *C* sunt fiii lui *A*, iar *B* este fratele lui *C*.

Ierarhizarea nodurilor se poate concretiza prin diferite tipuri de referinţe, în general folosindu-se *referinţe descendente (părinte, fiu)*, dar putându-se folosi şi *referinţe ascendente (fiu, părinte)* sau *orizontale (frate, frate)*.

Dacă numărul maxim de referinţe ale unui nod este *n*, atunci un element corespunzător unui nod poate fi reprezentat astfel:

Informaţie utilă	Leg_1	Leg_2	...	Leg_n

Leg_i conţine referinţa la cel de-al *i*-lea nod adiacent nodului dat. Absenţa unei legături se semnalează prin valoarea specială `nil`.

Dacă nu există o ierarhizare a nodurilor pe niveluri, atunci arborele se numeşte *neorientat*.

9.4.3. Reţeaua

Dacă nodurile unei mulţimi *M* nu se mai organizează pe niveluri, dându-se posibilitatea existenţei de legături între oricare două dintre ele, se obţine o structură de tip *reţea*. Fiecare nod *N* poate avea un număr oarecare de referinţe şi poate fi referit la rândul său de un număr oarecare de noduri.

9.5. Implementarea structurilor de date folosind alocare dinamică

În cazul în care anumitor variabile intenţionăm să le alocăm spaţiu în *heap*, acestea vor fi numite *variabile dinamice*. Tipul variabilelor dinamice poate fi orice tip recunoscut de limbajul de programare utilizat. Dar, atenţie: variabilele dinamice se declară prin intermediul referinţelor lor. Altfel spus, în secţiunea `var` (în Pascal) nu vom avea nici o declaraţie care să asocieze nume unei variabile dinamice, în schimb vom avea declaraţii ale variabilelor de tip referinţă (*pointer*ilor) care la rândul lor asigură accesul la variabila dinamică.

9.5.1. Variabile de tip referință (pointeri)

Variabila de tip referință (pointerul) este o variabilă alocată static. În limbajul Pascal tipul acestora este un tip special care nu este compatibil cu nici un alt tip de date. Regulile de prelucrare ale lor sunt bine precizate, deoarece valoarea unui pointer este o adresă din heap sau valoarea specială **nil**.

A. Declararea

Dacă dorim să declarăm o variabilă *ref* de tip referință care în timpul prelucrării va conține adresa unei variabile dinamice de tipul *tip_variabilă_dinamică*, declarația are următoarea formă generală:

```
type Tipref = ^tip_variabilă_dinamică;
     tip_variabilă_dinamică =...
var ref:Tipref;
```

Exemplu

```
type TipTablou=array[1..100,1..100] of Real;
     Tipref1=^Integer;
     Tipref2=^TipTablou;
var a,b:Tipref1;
    reftab:Tipref2;
```

Să observăm că în secțiunea **var** s-au declarat doar variabilele de tip referință. Declarația tipului `Tipref1` se „citeşte" în felul următor: pointerul de tipul `Tipref1` va conține adresa unei variabile de tip `Integer` (alocată în heap).

Tipurile pointerilor sunt diferite în funcție de tipul variabilei dinamice referite. De exemplu, pointerii `a` şi `reftab` din exemplul precedent sunt de tipuri diferite, iar `a` şi `b` au acelaşi tip.

B. Atribuirea

Unei variabile de tip pointer îi putem atribui valoarea unui alt pointer care are acelaşi tip sau putem să-i atribuim valoarea **nil**. De exemplu: `a:=b; b:=`**`nil`**.

Amintim că pointerii pot primi valoare şi altfel, în momentul creării unei variabile dinamice în heap, prin intermediul procedurii predefinite `New(ref)`, unde `ref` este o variabilă de tip referință.

C. Compararea pointerilor

Două variabile de tip referință pot fi comparate cu ajutorul operatorilor relaționali = (egal) şi <> (diferit). Rezultatul este o constantă logică (*adevărat* sau *fals*).

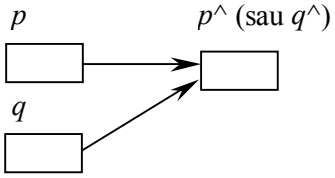
Operații, precum citirea şi scrierea unui pointer nu sunt posibile în limbajul Pascal. De asemenea, în Pascal nu putem avea operand pointer într-o expresie aritmetică.

9.5.2. Variabile dinamice

A. Referirea variabilei dinamice prin intermediul pointerului

Variabila dinamică, neavând nume propriu, se referă folosind variabila de tip pointer într-o construcție de forma ^*variabilă_de_tip_referință*.

La un moment dat, în timpul prelucrării, un pointer poate referi o singură variabilă dinamică, în schimb o variabilă dinamică poate fi referită de mai mulți pointeri. În figura următoare avem o variabilă dinamică referită atât de *p* cât şi de *q*, ceea ce înseamnă că variabila dinamică poate fi „numită” atât *p*^ cât şi *q*^, deoarece valorile acestora sunt egale şi conțin adresa variabilei dinamice.



B. Crearea unei variabile dinamice

Am amintit deja că spațiul necesar unei variabile dinamice se alocă prin intermediul procedurii predefinite `New(ref)`. Această alocare are loc în timpul execuției programului. În urma apelului mai întâi se caută un spațiu liber în heap, suficient de mare pentru a cuprinde o variabilă dinamică de tipul specificat în declararea variabilei `ref`. Dacă un astfel de spațiu liber există, atunci adresa de început al acestuia se reține în variabila `ref` şi se trece, împreună cu lungimea spațiului alocat, în lista adreselor ocupate. Zona respectivă poate fi ocupată de valoarea variabilei dinamice `ref^`. Valoarea variabilei dinamice poate fi citită sau calculată similar variabilelor statice, cu condiția să ne referim la ea cu construcția `ref^`. Dacă un astfel de spațiu nu există (heapul este ocupat în întregime sau este mult prea fragmentat), execuția programului se întrerupe cu mesajul `Heap overflow error`.

În astfel de situații mai întâi vom verifica dimensiunea structurilor de date, deoarece este posibil că într-adevăr este nevoie de mai mult spațiu. În acest caz se poate încerca mărirea dimensiunii heapului din mediul Borland Pascal-ului până la valoarea maximă de 655360 B. Dacă mesajul de eroare nu se justifică prin dimensiunea datelor alocate în heap, trebuie să căutăm greşeala în algoritm sau în implementarea lui.

C. Anularea unei variabile dinamice

Prin anulare înțelegem operația prin care eliberăm spațiul alocat unei variabile dinamice. Conținutul acestui spațiu nu se şterge fizic, în schimb la o eventuală alocare nouă este foarte probabil că în acel spațiu se rezervă loc altor variabile dinamice, ca atare valorile respective se vor suprascrie cu valorile variabilelor alocate ultima dată.

Procedura predefinită cu care realizăm eliberarea unui spațiu alocat unei variabile dinamice referite de pointerul `ref` este `Dispose(ref)`. Dacă valoarea pointerului `ref` este **`nil`**, apelul procedurii va conduce la mesajul de eroare `Invalid pointer operation`.

În limbajul Pascal există încă o modalitate de a elibera spațiul alocat în heap. Dacă marcăm un „moment" în timpul executării programului cu procedura `Mark(point)` şi mai târziu apelăm procedura `Release(point)`, putem elibera întreg spațiul alocat de către program între momentul marcării şi apelul procedurii `Release(point)`. Aici pointerul `point` este de tip `Pointer`, care este un tip referință predefinit în unit-ul *System* al mediului Borland Pascal. Acest tip este compatibil cu toate tipurile referință declarate în program, deci procedura `Release(point)` va elibera spațiile ocupate din heap, indiferent de tipul pointerilor utilizați.

În cazul unor incertitudini pot fi utile funcțiile `MemAvail` şi `MaxAvail`. Prima returnează dimensiunea totală a spațiului liber din heap, a doua returnează dimensiunea spațiului liber contiguu. Această valoare diferă de prima datorită faptului că în urma diferitelor alocări şi eliberări de spațiu, heapul se poate „fărâmița". Astfel putem afla dimensiunea maximă a unei variabile dinamice care se poate aloca într-un moment dat.

9.5.3. Implementarea dinamică a stivelor

A. Declararea stivei

Să considerăm o stivă în care elementele sunt numere întregi. Declararea o realizăm recursiv, astfel încât să se oglindească înlănțuirea. Variabilele dinamice vor avea tipul **`record`**, deoarece în fiecare vom reține valoarea numărului (informația propriu-zisă) şi valoarea adresei (informația de legătură) următorului element de pe stivă (cel care se află imediat sub primul element).

```
type Stiva=^Element;                                  { tipul pointerului }
     Element=record                                   { tipul variabilei dinamice }
                nr:Integer;
                leg:Stiva
              end;
var cap,p:Stiva;          { cap va fi pointerul către primul element din stivă }
```

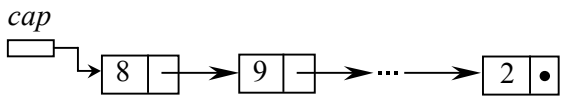
B. Inserarea unui element nou într-o stivă alocată dinamic

În stivă orice adăugare se face la început, astfel încât pointerul `cap` va indica întotdeauna elementul care „se vede", adică ultimul inserat, respectiv primul care se poate elimina. Secvența de program cu care realizăm o astfel de adăugare este următoarea:

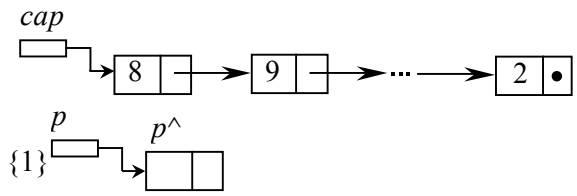
```
procedure InserareStiva(var cap:Stiva; ce:Integer);
          { ce conține valoarea care se adaugă în stiva referită de pointerul cap }
var p:Stiva;                                   { p este un pointer ajutător }
begin
  New(p);                                                          { 1 }
  p^.nr:=ce;                                                       { 2 }
  p^.leg:=cap;                                                     { 3 }
  cap:=p                                                           { 4 }
end;
```

Exemplu

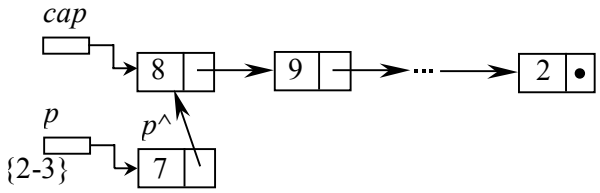
Să presupunem că în momentul apelării subprogramului de inserare stiva are următorul conținut:



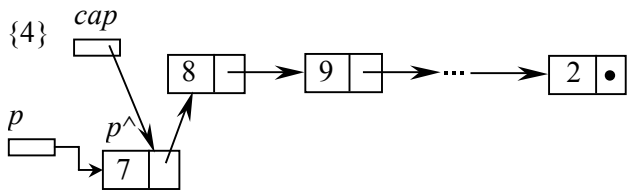
În urma apelului procedurii de inserare mai întâi se alocă spațiu variabilei dinamice *p*^ (instrucțiunea marcată cu {1}). Deocamdată nici un câmp de-al său nu a primit nici o valoare:



La pasul {2} şi {3} se atribuie valori câmpurilor variabilei dinamice *p*^. Fie *ce* = 7. Deoarece noul element se inserează în vârful stivei, câmpul de legătură a variabile dinamice *p*^ va indica acelaşi element pe care îl indică pointerul *cap*.



Pasul {4} finalizează operația prin actualizarea pointerului *cap*, astfel încât acesta să indice ultimul element inserat:



C. Crearea unei stive alocate dinamic

Dacă dorim să creăm o stivă, vom apela în mod repetat subprogramul de inserare într-o instrucțiune repetitivă. Capul stivei trebuie inițializat anterior apelului cu valoarea **`nil`**.

D. Eliminarea unui element dintr-o stivă alocată dinamic

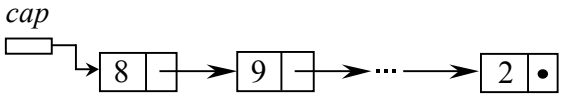
Dintr-o stivă se va elimina întotdeauna elementul referit de capul stivei. Următorul subprogram realizează eliminarea acestui element şi eliberează spațiul alocat lui. În parametrul *ce* vom păstra informația propriu-zisă din elementul eliminat din stivă.

Este important ca un astfel de subprogram să se apeleze doar după ce în prealabil ne-am asigurat că stiva nu este vidă.

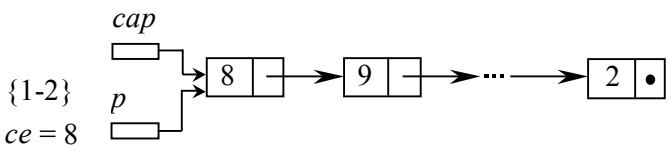
```
procedure EliminareStiva(var cap:Stiva; var ce:Integer);
     { ce va conține valoarea care a existat în vârful stivei referite de pointerul cap }
var p:Stiva;                                        { p este un pointer ajutător }
begin
  ce:=cap^.nr;                                                          { 1 }
  p:=cap;                                                               { 2 }
  cap:=cap^.leg;                                                        { 3 }
  Dispose(p)                                                            { 4 }
end;
```

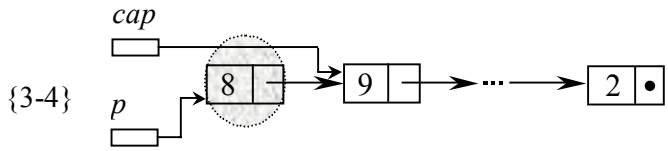
Exemplu

Fie stiva:



Să presupunem că din aceasta dorim să eliminăm un element. Paşii subprogramului de eliminare se pot urmări în figurile următoare:





Spațiul haşurat rămâne deocamdată „ocupat” de elementul aflat acolo, dar adresa acestuia se returnează sistemului de gestiune a heap-ului care la o cerere viitoare poate aloca acest spațiu unei alte variabile. Capul stivei acum indică elementul care conține numărul 9.

Observație

Operația de căutare, de inserare sau de eliminare în/din altă parte decât în/din vârful stivei nu au sens, deoarece contravin definiției acestei structuri de date. De asemenea, afişarea conținutului stivei se consideră ca fiind o prelucrare în care se va afişa elementul din vârf, apoi acesta se elimină, astfel se va vedea următorul element şi se va afişa etc.

9.5.4. Implementarea dinamică a cozilor

A. Declararea cozii

Declararea cozii este identică cu cea a stivei, dar în afară de pointerul care referă capul cozii vom avea nevoie de încă unul care să refere ultimul element, deoarece inserarea se va face după ultimul element. Eliminarea din coadă este identică cu eliminarea din stivă, deoarece şi din cozi se elimină elementul din vârf.

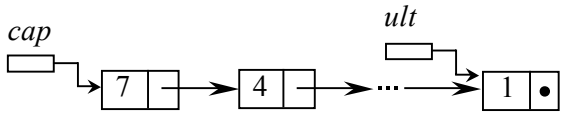
B. Inserarea unui element nou într-o coadă alocată dinamic

În coadă orice adăugare se face după ultimul element inserat, astfel încât pointerul *ult* va indica întotdeauna ultimul element inserat. Totodată pointerul *cap* va indica primul element care se poate elimina. Secvența de program corespunzătoare este următoarea:

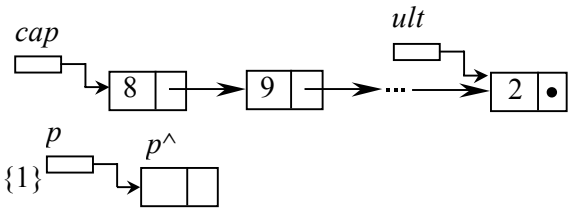
```
procedure InserareCoada(var cap,ult:Coada; ce:Integer);
          { ce conține valoarea care se adaugă în coada referită de pointerul cap }
var p:Coada;                                   { p este un pointer ajutător }
begin
  New(p);                                                          { 1 }
  p^.nr:=ce;                                                       { 2 }
  p^.leg:=nil;                                                     { 3 }
  ult^.leg:=p                                                      { 4 }
  ult:=p                                                           { 5 }
end;
```

Exemplu

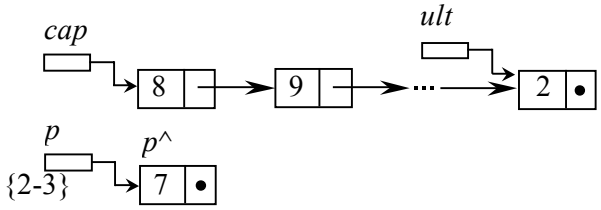
Să presupunem că în momentul apelării subprogramului de inserare coada are următorul conținut:



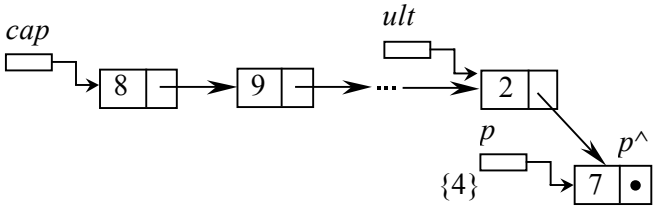
În urma apelului procedurii de inserare mai întâi {1} se alocă spațiu variabilei dinamice *p*^. Deocamdată nici un câmp de-al său nu a primit nici o valoare.



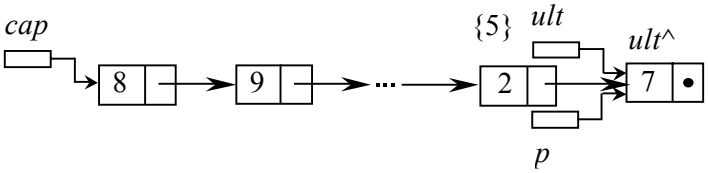
La paşii {2-3} se atribuie valori câmpurilor variabilei dinamice *p*^. Fie *ce* = 7. Deoarece noul element se inserează la sfârşitul cozii, câmpul de legătură a variabile dinamice *p*^ va primi valoarea **`nil`**.



La pasul {4} se leagă variabila dinamică *p*^ după ultimul element al cozii:



În final se actualizează pointerul *ult* astfel încât să indice ultimul element inserat:



C. Crearea unei cozi alocate dinamic

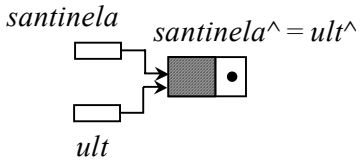
Dacă dorim să creăm o coadă, vom apela în mod repetat subprogramul de inserare într-o instrucțiune repetitivă. Pentru a realiza fiecare inserare, deci şi inserarea primului element, avem nevoie de un *ultim element*, după care se realizează inserarea. Rezultă că inițializarea cozii înseamnă crearea mai întâi a unui element care să fie şi primul şi ultimul.

```
procedure InitCoada(var cap,ult:Coada; ce:Integer);
        { creăm o coadă formată dintr-un singur element care conține valoarea ce }
begin
  New(cap);
  cap^.nr:=ce;
  cap^.leg:=nil;
  ult:=cap
end;
```

Faptul că primul element trebuie inserat altfel decât restul elementelor constituie un oarecare inconvenient care poate fi înlăturat prin introducerea în coadă a unui element fictiv, numit frecvent *santinelă*. Acest element se creează înainte de apelul subprogramului de creare propriu-zisă a cozii:

```
...
  New(santinela);
  santinela^.leg:=nil;
  ult:=santinela
...
```

Coada vidă astfel creată conține doar acest element fictiv:



Crearea cozii în acest caz se realizează cu subprogramul `InserareCoada` apelat cu parametri actuali `santinela`, `ult` şi `ce`.

D. Eliminarea unui element dintr-o coadă alocată dinamic

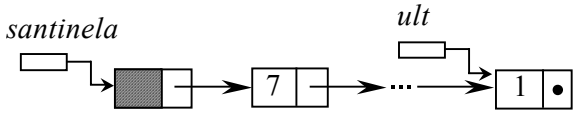
Dintr-o coadă se va elimina întotdeauna elementul referit de capul cozii. Subprogramul care realizează eliminarea acestui element şi eliberează spațiul alocat lui este identic cu subprogramul de eliminare din stivă.

Eliminarea elementului în cazul în care am lucrat cu santinelă, trebuie să țină cont de faptul că primul element din coadă este santinela. În concluzie se va elimina variabila dinamică *santinela^.leg^*. Procedura `Eliminare2Coada` va fi apelată doar dacă *santinela^.leg* nu este **`nil`**, adică în cazul în care avem cel puțin un element în coadă în afara santinelei.

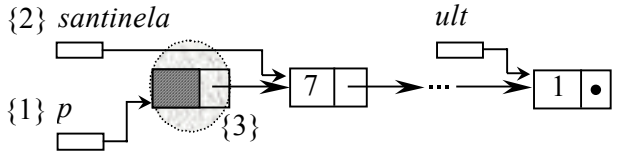
```
procedure Eliminare2Coada(santinela,ult,ce);
var p:Coada;
begin
  p:=santinela;                                              { 1 }
                              { elementul care trebuie eliminat devine santinelă }
  santinela:=santinela^.urm;                                 { 2 }
  Dispose(p)                                  { 3 – se elimină santinela veche }
end;
```

Exemplu

Fie coada:



Să presupunem că din aceasta dorim să eliminăm un element. Paşii subprogramului de eliminare se pot urmări în figura următoare:



Observație

Operații de căutare, inserare sau eliminare în/din altă parte decât în/de la vârful/ sfârşitul cozii nu au sens, deoarece contravin definiției acestei structuri de date. De asemenea, afişarea conținutului cozii se consideră ca fiind o prelucrare în care se va afişa elementul din vârf, apoi acesta se elimină, astfel se va vedea următorul element, care se va afişa etc.

9.5.5. Implementarea listelor liniare simplu înlănțuite

O listă simplu înlănțuită poate fi creată similar stivelor sau cozilor, dar pot apărea şi situații în care inserarea trebuie făcută undeva în interiorul listei între două elemente existente. La fel, şi eliminarea se poate realiza de oriunde din listă. Aceste operații au

sens dacă lista conține elementele într-o anumită ordine, unde se poate căuta, de exemplu, elementul după care (sau eventual în fața căruia) vrem să inserăm elementul nou. De asemenea, eliminările se realizează nu neapărat dintr-un capăt al listei, ci se poate elimina elementul având o anumită proprietate, eventual cel din fața unui element sau aflat după un anumit element.

A. Căutarea unui element într-o listă simplu înlănțuită, ordonată crescător

Căutarea în liste alocate dinamic se realizează secvențial, chiar dacă acestea sunt ordonate. Dacă lista este neordonată algoritmul se termină fie în momentul în care am găsit elementul căutat, fie după ce am verificat şi ultimul element:

```
function Cauta(prim:list; cautat:Integer):Boolean;
var p:list;                                        { list: tipul referință }
    gasit:Boolean;
begin
  p:=prim;                      { primul element este indicat de pointerul prim }
  gasit:=false;                                  { gasit: variabilă auxiliară }
  while not gasit and (p <> nil) do
    if p^.info = cautat then { se caută elementul având valoarea cautat }
      gasit:=true
    else
      p:=p^.leg;
  Cauta:=gasit
end;
```

Dacă avem o listă în care există un câmp *cheie*, valorile căruia sunt ordonate crescător sau descrescător în cadrul succesiunii elementelor, căutarea se va face de asemenea secvențial, dar algoritmul se termină fie când am găsit elementul căutat, fie când ne-am dat seama că am depăşit locul unde ar fi trebuit să se afle acesta, deci nu mai are rost să-l căutăm. În acest caz, algoritmul de căutare este următorul:

```
function Cauta(prim:list; cautat:Integer):Boolean;
var p:list;
    gasit:Boolean;
begin
  p:=prim;
  gasit:=false;
  while not gasit and (p <> nil) do
    if p^.info=cautat then gasit:=true
    else
      if p^.info < cautat then p:=p^.leg
      else Break;                          { echivalent aici cu p:=nil; }
  Cauta:=gasit
end;
```

B. Inserarea unui element într-o listă simplu înlănțuită, în fața elementului referit de pointerul *p*

Să considerăm cazul în care dorim să construim o listă în care elementele să fie ordonate crescător după câmpul cheie *info*. Având în vedere că pe parcursul inserării elementelor va trebui să căutăm locul fiecăruia, vom avea nevoie de un subprogram care să returneze, de exemplu, valoarea pointerului *p*, care referă elementul în fața căruia dorim să inserăm noul element. Locul acestui element nou poate fi în fața primului element, în fața unuia în interiorul listei, dar este posibil ca noul element să trebuiască pus ultimul în listă, caz în care valoarea lui *p* este **nil**. Observăm că în aceste condiții, pe parcursul creării listei vom fi nevoiți să ținem evidența atât a pointerului care indică primul element, cât şi referința la ultimul.

Algoritmul de căutare a elementului în fața căruia se va insera noul element se poate implementa în Pascal cu funcția `Cauta`. Variabila auxiliară *găsit* ne ajută să evităm o expresie logică (în instrucțiunea **while**) în care să verificăm conținutul unui câmp dintr-o variabilă dinamică inexistentă (dacă *p* este egal cu **nil**, *p*^.*info* nu există).

```
function Cauta(prim:list; cautat:Integer):list;
     { returnează pointerul p care indică elementul în fața căruia se face inserarea }
var p:list;
    gasit:Boolean;
begin
  p:=prim;
  gasit:=false;
  while not gasit and (p <> nil) do
    if cautat>=p^.info then
      p:=p^.leg
    else
      gasit:=true;
  Cauta:=p
end;
```

Inserarea elementului se realizează în funcție de valoarea pointerului returnat de funcția `Cauta`.

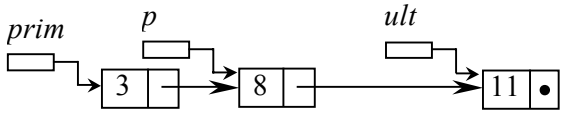
1. Dacă acesta este egal cu *prim*, noul element se inserează în vârful listei (inserare ca în stive) şi pointerul care îl indică devine noul *prim*.
2. Dacă valoarea pointerului returnat este egal cu **nil**, noul element se inserează la sfârşitul listei (inserare ca în cozi) şi pointerul care îl indică devine noul *ultim*.
3. Dacă valoarea pointerului returnat nu este egal nici cu *prim*, nici cu *ultim*, avem o inserare între două elemente în interiorul listei.

Problema care apare se datorează faptului că aparent degeaba cunoaştem pointerul la elementul în *faţa* căruia dorim să inserăm noul element, deoarece nu mai cunoaştem pe cel *după* care va urma acesta şi în concluzie, nu avem cum să legăm elementul nou de prima parte a listei. În acest caz inserarea se realizează cu următoarea secvenţă de instrucţiuni:

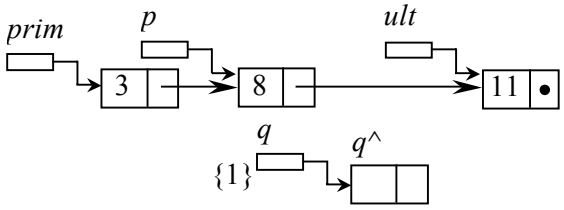
```
...
  New(q);                                          {1}
  q^:=p^;                                          {2}
  p^.info:=ce;                                     {3}
  p^.leg:=q;                                       {4}
  if p=ultim then                  { 5 – dacă am inserat în faţa lui ultim }
    ultim:=q;
...
```

Exemplu

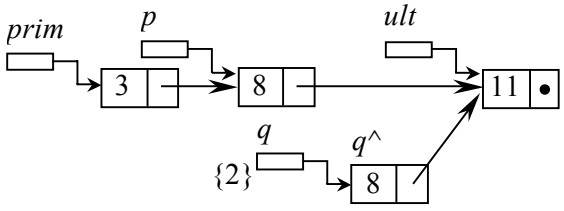
Fie lista în momentul în care vrem să inserăm valoarea *ce* = 5. Funcţia `Cauta` a returnat valoarea pointerului *p* care indică elementul în faţa căruia vrem să-l punem (care are valoarea câmpului *info* egală cu 8).



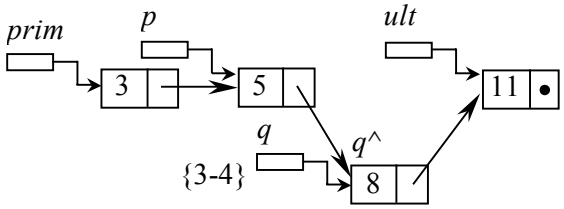
Să urmărim modul în care acţionează instrucţiunile din secvenţa de mai sus. La primul pas se alocă spaţiu pentru o variabilă dinamică nouă, care se va referi de pointerul *q*:



La pasul următor copiem conţinutul variabilei dinamice referite de *p* în variabila indicată de *q* (se copiază toate câmpurile):



Astfel am „eliberat” elementul indicat de *p* (am „salvat” valoarea 8 în elementul indicat de *q*) şi atribuim câmpului *info* din variabila dinamică *p^* valoarea 5. De asemenea, legăm elementul indicat de *p* de elementul referit de *q*:



Pasul {5} este necesar, deoarece se observă că pe parcursul procesului descris am mutat elementul în faţa căruia am realizat inserarea din locaţia unde a fost păstrată într-o locaţie nouă. Dacă acest element, a fost ultimul, (deci pointerul *ultim* conţinea adresa lui), acum, mutându-l, trebuie să actualizăm şi valoarea pointerului *ultim*. Aceasta ar fi fost cazul, dacă am fi dorit să inserăm un element nou având valoarea 10. Locul acestuia ar fi fost în faţa elementului având valoarea câmpului *info* egal cu 11. Creând un element nou pentru 11, practic s-ar fi creat un nou ultim element.

C. Inserarea unui element într-o listă simplu înlănţuită, după elementul referit de *p*

Dacă avem o funcţie `Cauta` care returnează valoarea pointerului *după* care se doreşte inserarea, secvenţa de program este mai simplă, în schimb căutarea este uşor mai complicată. De exemplu, dacă primul element din listă conţine valoarea 1 şi vrem să găsim locul lui 0, nu există acel element *după* care 0 trebuie inserat. În concluzie acest caz îl tratăm separat şi atribuim pointerului returnat valoarea **`nil`**, care va însemna că noul element va fi noul *prim*.

În structura repetitivă vom sări peste elementul indicat de pointerul *prim*, deoarece acesta a fost deja testat. Astfel vom verifica valoarea variabilei *p^.leg^.info*. În aceste condiţii va trebui să asigurăm ieşirea din structura repetitivă în momentul în care am verificat şi ultimul element, şi să nu verificăm valoarea câmpului *info* a unui element inexistent. (Bineînţeles, există şi alte modalităţi de a rezolva problema.)

```
function Cauta(prim:list; cautat:Integer):list;
        { returnează pointerul p care indică elementul după care se face inserarea }
var p:list;
begin
  if prim^.info > cautat then
    p:=nil                                          { vom avea un nou prim }
  else begin
    p:=prim;                          { prim va fi „sărit”; l-am verificat deja }
    while p^.leg^.info<cautat do begin
      p:=p^.leg;
```

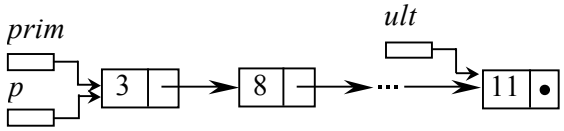
```
      if p^.leg = nil then                    { p^.leg^.info nu există }
        Break
    end
  end;
  Cauta:=p
end;
```

Secvența de program cu care inserăm un element între două elemente din listă, atunci când se cunoaşte pointerul elementului după care se doreşte inserarea:

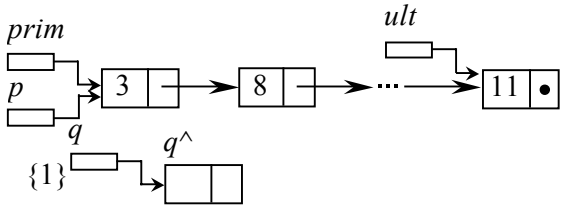
```
...
  New(q);                                                    {1}
  q^.info:=ce;                                               {2}
  q^.leg:=p^.leg;                                            {3}
  p^.leg:=q                                                  {4}
...
```

Exemplu

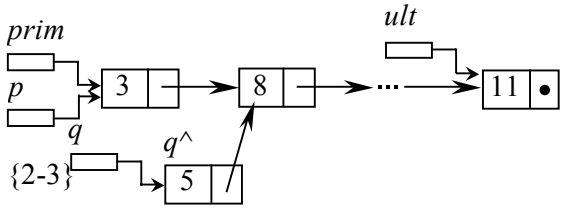
Fie lista în momentul în care vrem să inserăm valoarea *ce* = 5. Funcția `Cauta` a returnat valoarea pointerului *p* care indică elementul după care vrem să-l punem (care are valoarea câmpului *info* egală cu 3).



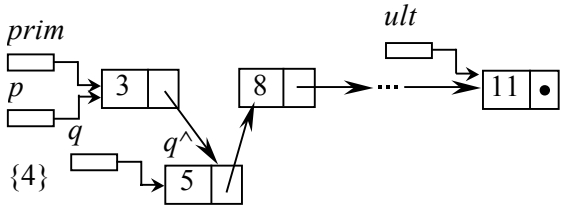
Să urmărim modul în care acționează instrucțiunile din secvența de mai sus. La primul pas se alocă spațiu pentru o variabilă dinamică nouă, care se va referi de pointerul *q*:



Acum copiem valoarea variabilei *ce* în variabila indicată de *q*, iar câmpul de legătură al acestui element va indica elementul care urmează după cel indicat de *p*:



Acum trebuie să legăm elementul nou creat de prima parte a listei, adică să modificăm câmpul de legătură a variabilei dinamice *p*^:



D. Eliminarea elementului referit de pointerul *p*

Presupunem că pointerul *p* conține referința la elementul pe care trebuie să-l eliminăm din listă. O astfel de eliminare se poate realiza în felul următor:

```
procedure Sterge(var prim,ultim:list; p:list);
var q:list;
begin
  if p <> ultim then begin              { dacă suntem în interiorul listei }
    q:=p^.leg;                  { asigurăm accesul la elementul care va fi şters }
    p^:=p^.leg^;     { peste elementul care trebuie şters îl copiem pe următorul }
    if q = ultim then        { este posibil să ştergem de fapt ultimul element }
      ultim:=p;                                  { actualizăm pointerul ultim }
    Dispose(q)                                    { ştergem vechiul „următor” }
  end else begin
                        { dacă avem mai multe elemente, şi trebuie şters ultimul }
    if prim <> ultim then begin
      q:=prim;
      while q^.leg <> ultim do                  { îl căutăm pe penultimul }
        q:=q^.leg;
      q^.leg:=nil;         { actualizăm vechiul penultim care va fi noul ultim }
      ultim:=q;                                  { actualizăm pointerul ultim }
      Dispose(p)                                 { eliberăm spațiul referit de p }
    end else begin                             { avem un singur element în listă }
      Dispose(prim);
      prim:=nil;
    end
  end
end;
```

E. Traversarea listelor simplu înlănțuite

Traversarea se realizează cu un subprogram simplu, în care trecem de la element la element, urmând înlănțuirea, până la elementul care în câmpul de legătură are **nil**.

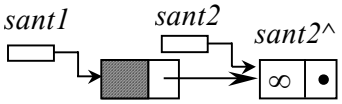
```
procedure Afiseaza(prim:list);
var p:list;
begin
  p:=prim;
  while p <> nil do begin
    WriteLn(p^.info);
    p:=p^.leg
  end
end;
```

9.5.6. Liste simplu înlănțuite implementate cu santinele

Am văzut că realizarea operațiilor de adăugare, respectiv eliminare se realizează anevoios datorită faptului că în funcție de poziția elementului, algoritmii se ramifică în funcție de o mulțime de cazuri. Dacă lista se implementează cu santinele (unul la început şi unul la sfârşit, operațiile se pot realiza mult mai uşor, deoarece „dispar" cazurile de inserare în fața primului element, respectiv după ultimul precum ştergerea ultimului sau penultimului.

A. Inserarea unui element într-o listă simplu înlănțuită, în fața elementului referit de pointerul *p*

În cazul unei implementări cu două santinele lista vidă are următoarea structură:

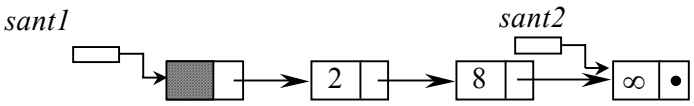


Presupunem că dorim să creăm o listă ordonată crescător. Lista vidă se creează cu secvența de instrucțiuni:

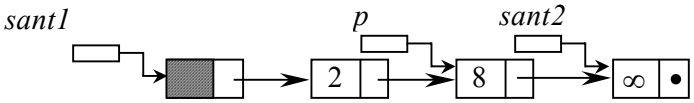
```
...
New(sant1); New(sant2);
sant2^.info:=MaxInt;     { o valoare mare, astfel încât celelalte să fie mai mici }
sant1^.leg:=sant2;
...
```

Exemplu

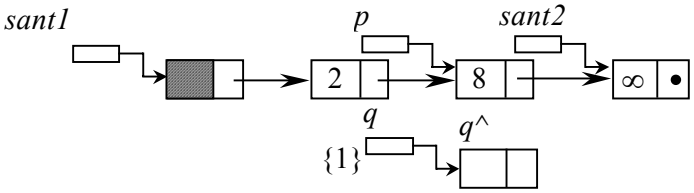
Fie lista:



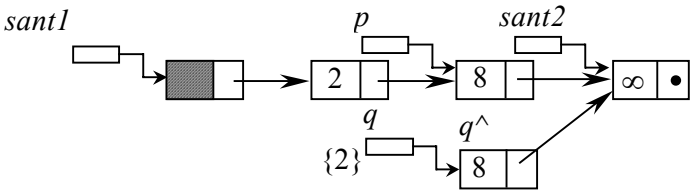
Dorim să inserăm un element nou pentru valoarea 5 şi avem la dispoziție un pointer *p* care indică elementul care în câmpul *info* are valoarea 8:



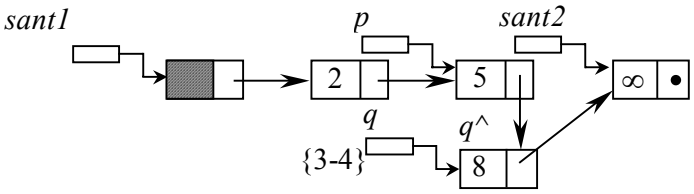
La pasul următor alocăm spațiu noului element:



Deoarece avem la dispoziție adresa unui element care este *după* locul în care dorim inserarea, mai întâi vom copia acest element în *q*^:



La ultimul pas introducem valoarea nouă în elementul *p*^ şi legăm noul element de prima parte a listei:



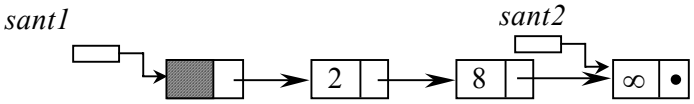
Folosind santinela, este posibil ca noul element să trebuiască inserat în fața santinelei *sant2*. În acest caz, în *q*^ vom copia variabila dinamică *sant2*^, deci va trebui să actualizăm valoarea acestuia {5}.

Secvența de program cu care realizăm inserarea noului element este următoarea:

```
...
New(q);                                                    {1}
p^:=q^;                                                    {2}
p^.info:=ce;                                               {3}
p^.leg:=q;                                                 {4}
if p=sant2 then sant2:=q;                                  {5}
...
```

B. Inserarea unui element într-o listă simplu înlănțuită, după elementul referit de *p*

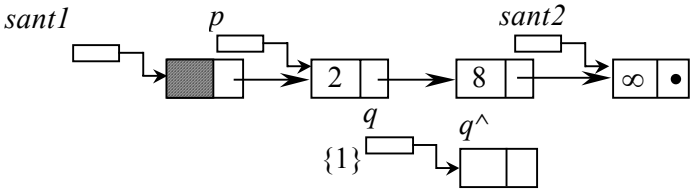
Fie o listă în care cheile sunt ordonate crescător, având următorul conținut:



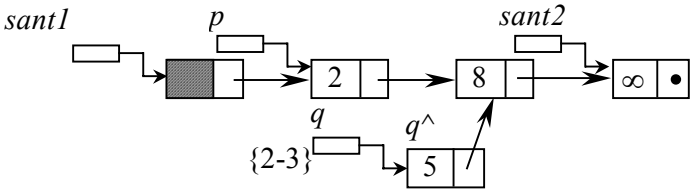
Dorim să inserăm un element nou având valoarea 5. Adăugăm elementul nou cu următoarea secvență de program:

```
...
New(q);                                    {1}
q^.info:=ce;                               {2}
q^.leg:=p^.leg;                            {3}
p^.leg:=q;                                 {4}
...
```

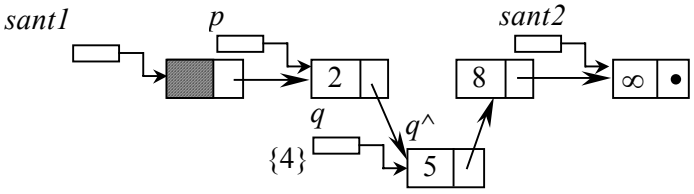
Să urmărim efectul secvenței de instrucțiuni pe figurile următoare. Pointerul *p* indică elementul care conține valoarea 2, după care dorim să inserăm valoarea 5. La primul pas se alocă spațiu pentru o variabilă dinamică nouă *q*^:



Urmează încărcarea câmpurilor variabilei *q*^:



La pasul următor legăm prima parte a listei de elementul nou creat:



D. Eliminarea elementului referit de pointerul *p*

Atunci când vrem să eliminăm elementul *p*^, practic trebuie să legăm elementul aflat în fața lui de cel care urmează după el. Vom proceda similar cu adăugarea, respectiv cu eliminarea din liste simplu înlănțuite şi anume copiem în *p*^, elementul *p*^.*leg*^ şi eliberăm zona de memorie alocată acestuia din urmă:

```
...
q:=p^.leg;                          { păstrăm adresa zonei care se va elibera }
p^:=q^;                { copiem elementul următor în elementul care trebuie eliminat }
if q=sant2 then  { dacă am copiat sant2, actualizăm valoarea acestei santinele }
  sant2:=p;
Dispose(q);                                          { eliberăm spațiul ocupat }
...
```

E. Traversarea listelor simplu înlănțuite

Traversarea se realizează ca în cazul listelor implementate fără santinele, dar în acest caz nu se „traversează” santinelele:

```
procedure Afiseaza(sant1,sant2:list);
var p:list;
begin
  p:=sant1^leg;
  while p <> sant2 do begin
    WriteLn(p^.info);
    p:=p^.leg
  end
end;
```

9.5.7. Liste dublu înlănțuite

Dacă lista este astfel construită încât înlănțuirea este păstrată în informații de legătură care rețin atât adresa elementului precedent, cât şi al succesorului, avem liste dublu înlănțuite. Din nou accentuăm avantajul utilizării santinelelor, astfel nemaifiind necesară tratarea separată a cazurilor când inserarea, respectiv eliminarea se face la începutul sau la sfârşitul listei.

Inițializarea listei începe cu crearea unei liste vide care conține doar cele două santinele:

```
...
New(sant1);
New(sant2);
sant2^.pred:=sant1;
sant1^.urm:=sant2;
...
```

A. Declararea listei dublu înlănțuite

În declararea listei dublu înlănțuite, pe lângă câmpurile care cuprind informațiile propriu-zise, vom specifica două câmpuri de legătură: a predecesorului şi a succesorului (*pred* şi *urm*).

B. Inserarea unui element nou într-o listă dublu înlănțuită

Să presupunem că în pointerul *p* avem la dispoziție adresa elementului în fața căruia trebuie să inserăm noul element.

Secvența de program este următoarea:

```
...
New(q);
q^.info:=...
q^.urm:=p;
q^.pred:=p^.pred;
p^.pred^.urm:=q;
p^.pred:=q;
...
```

C. Eliminarea unui element nou într-o listă dublu înlănțuită

Având la dispoziție atât legătura către elementul predecesor cât şi către cel următor, eliminarea elementului referit de pointerul *p* se realizează simplu:

```
...
p^.pred^.urm:=p^.urm;
p^.urm^.pred:=p^.pred;
Dispose(p);
...
```

D. Traversarea listelor dublu înlănțuite

Traversarea listelor dublu înlănțuite se realizează similar cu cea a listelor simplu înlănțuite, cu precizarea că acestea se pot parcurge de la primul element spre ultimul, urmând înlănțuirea din câmpul *urm*, sau de la ultimul spre primul, conform legăturii din câmpul *pred*.

9.5.8. Liste circulare

Dacă într-o listă simplu sau dublu înlănțuită câmpul de legătură a ultimului element indică primul, avem listă circulară. În cazul listelor dublu înlănțuite, în plus, câmpul *pred* al primului element va indica ultimul element.

9.6. Structuri arborescente

9.6.1 Reprezentarea structurilor arborescente

A. Reprezentarea arborilor binari

Arborele binar este o structură de date care permite o descriere recursivă. Am putea da o descriere intuitivă în felul următor:

```
type arbore = nil or
              record
                info:T;
                stâng,drept:arbore
              end;
```

adică un arbore este fie vid, fie este format dintr-un nod care conţine informaţii de tip `T` şi din două alte componente care la rândul lor sunt arbori şi anume subarborele stâng, respectiv cel drept.

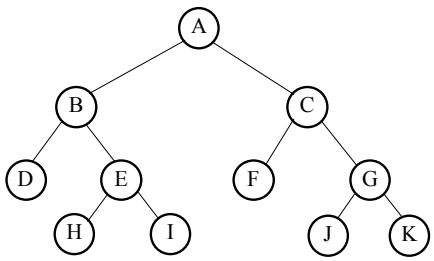
O astfel de descriere recursivă este incomodă în cadrul oricărui limbaj de programare. În limbajele de tip Pascal descrierea arborilor se realizează în felul următor:

```
type ref=^varf;
     varf=record
            info:T;
            stâng,drept:ref
          end;
```

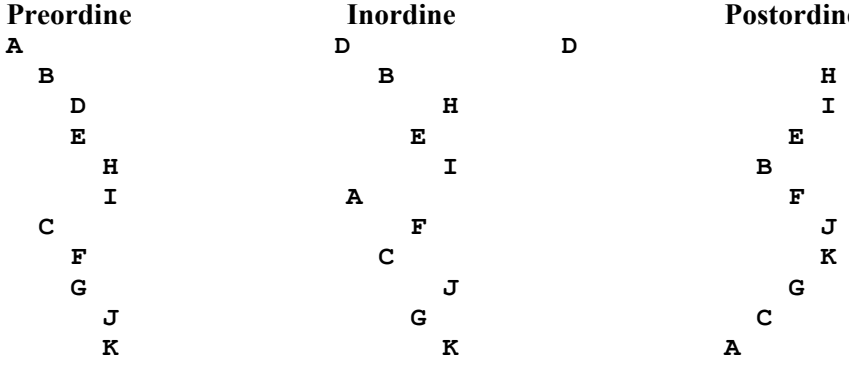
Această descriere corespunde reprezentării arborelui cu referinţe descendente (către fiii săi).

B. Reprezentarea arborilor pe suportul de ieşire (vizualizarea)

În practică, pot să apară situaţii în care o structură arborescentă trebuie vizualizată pe un suport de ieşire. De regulă se preferă afişarea cu indentare. Fie arborele din figura următoare:



Afişând acest arbore cu indentare, îl obţinem sub una din următoarele trei forme:



C. Reprezentarea pe suportul de intrare (citirea arborelui)

Introducerea unei structuri arborescente în memorie este o problemă care se rezolvă în general mai uşor decât vizualizarea. Dacă arborele este *binar*, una dintre posibilităţi este aceea de a citi informaţiile ataşate nodurilor în preordine; lipsa subarborelui stâng, respectiv drept se semnalează printr-o valoare specială, aleasă în funcţie de tipul cheilor ataşate nodurilor.

În scopul introducerii unui arbore *oarecare* în memorie, se pot citi informaţiile ataşate nodurilor în postordine, pentru fiecare nod aceste informaţii fiind însoţite de numărul descendenţilor nodului respectiv.

9.6.2. Operaţii elementare pe arbori

Se consideră operaţii elementare pe structuri arborescente: *construirea*, *traversarea arborelui* şi *căutarea*, *inserarea* şi *ştergerea* unor elemente într-o structură dată. Dat fiind faptul că în scopul sortării unei mulţimi s-au dovedit a fi utile structurile arborescente, în cele ce urmează vom aminti şi operaţia de *sortare*.

A. Construirea unui arbore binar total echilibrat

Un arbore binar total echilibrat are toate vârfurile terminale pe ultimele două niveluri astfel încât, pentru orice nod, numărul nodurilor din subarborele stâng diferă cel mult cu unu de numărul nodurilor din subarborele drept.

Procedeul de construire este următorul:

1. un vârf va fi rădăcină;
2. se pregătesc $ns = [n/2]$ vârfuri pentru subarborele stâng şi se trece la construirea lui (pasul 1);
3. se pregătesc $nd = n - ns - 1$ vârfuri pentru subarborele drept şi se trece la construirea lui (pasul 1);

Funcția `Arbore(n)` realizează construirea unui arbore binar total echilibrat (funcția returnează pointer la rădăcina arborelui).

```
function Arbore(n):ref;                                  { generare arbore }
var varfnou:ref;
    x,ns,nd:Integer;
begin
  if n = 0 then
    Arbore:=nil                                  { nici un nod ⇒ arbore vid }
  else begin
    ns:=n div 2                  { numărul nodurilor din subarborele stâng }
    nd:=n-ns-1                                    { respectiv din cel drept }
    New(varfnou)                                    { creare nod curent }
    with varfnou^ do begin
    Read(f,cod)                  { informația propriu-zisă pentru nodul nou }
    stang:=arbore(ns)                          { construire subarbore stâng }
    drept:=arbore(nd)                                 { respectiv drept }
  end;
  Arbore:=varfnou
  end
end;
```

B. Traversarea arborilor binari

Traversarea unui arbore binar constă în parcurgerea pe rând a vârfurilor arborelui şi punerea în evidență a fiecărui nod la numai una din atingerile sale (spunem că „vizităm" vârful respectiv, prin aceasta înțelegând efectuarea unor prelucrări ale informațiilor ataşate vârfului, prelucrări care pot fi oricând inserate în algoritmul traversării). Există două posibilități de parcurgere a arborilor: *în adâncime* şi *în lățime*. Cea mai folosită este cea în adâncime, care la rândul ei poate fi de trei feluri, cunoscute sub denumirile de *parcurgere în inordine, preordine şi postordine.*

Aceste denumiri sugerează poziția rădăcinii (în traversare) față de cei doi descendenți (respectiv între ei, înaintea lor sau după ei). Algoritmul de traversare este des utilizat, deoarece prelucrările care vizează toate datele unei structuri arborescente necesită un „drum" prin structură, astfel încât fiecare nod să fie vizitat o singură dată.

1. Traversarea în adâncime

Dat fiind faptul că cele trei tipuri de traversări în adâncime se realizează pe baza a trei algoritmi asemănători, vom prezenta doar algoritmul recursiv corespunzător traversării în preordine:

1. se vizitează rădăcina;
2. se traversează subarborele stâng;
3. se traversează subarborele drept.

Implementarea în Pascal a subalgoritmilor de traversare o prezentăm realizând totodată şi o afişare a arborelui binar cu indentare pentru cele trei tipuri de parcurgere. Procedura `Traversare(modtrav)` realizează traversarea arborelui binar în toate cele trei moduri de traversare în adâncime. În limbajul Pascal, parametrul `modtrav` este de tip enumerat şi are valorile `preordine`, `inordine` şi `postordine`.

```
procedure Afiseaza(nod:ref; nivel:Integer);
begin                                          { tipărirea nodului cu indentare }
  while nivel > 0 do begin          { indentare în funcţie de nivelul nodului }
    Write(g,'   ');
    Dec(nivel)
  end;
  WriteLn(g,nod^.cod)              { informaţia propriu-zisă ataşată nodului }
end;

procedure Travers(modt:trav; rad:ref; nivel:Integer);
begin                                              { traversarea propriu-zisă }
  if rad <> nil then begin
    if modt = preordine then
      Afiseaza(rad,nivel);
    Travers(modt,rad^.stang,nivel+1);
    if modt = inordine then
      Afiseaza(rad,nivel);
    Travers(modt,rad^.drept,nivel+1);
    if modt = postordine then
      Afiseaza(rad,nivel)
  end
end;

procedure Traversare(modtrav:trav);              { traversare în adâncime }
begin
  Travers(modtrav,radacina,0)
end;
```

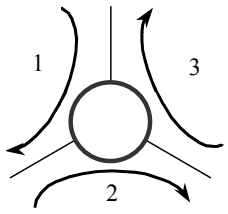
2. Traversarea nerecursivă

Evident, se poate obţine uşor o variantă de traversare nerecursivă, pornind de la o variantă recursivă şi ridicând recursivitatea printr-una din metodele cunoscute, de exemplu utilizând o stivă.

Orice traversare necesită *o zonă de memorie suplimentară* în care se păstrează anumite informaţii necesare traversării. Dacă această zonă creşte odată cu creşterea complexităţii structurii (cum este cazul stivei), apare pericolul de a nu putea traversa un arbore existent deja în memorie din cauza epuizării la un moment dat a memoriei disponibile.

Vom prezenta un algoritm care necesită pentru traversare o memorie suplimentară constantă, dar care foloseşte memorie suplimentară în fiecare nod al arborelui.

În subprogramul `Traversare_nerecursiva(modtrav)` arborele este considerat ca fiind reprezentat prin referinţele ascendente cât şi cele descendente ale sale şi în plus, în fiecare nod păstrându-se un contor care numără de câte ori s-a trecut deja prin nodul respectiv în decursul traversării (pentru a putea realiza mai multe traversări succesive, contorul va fi un numărător modulo 3, vezi figura).



În subprogramul `Traversare_nerecursiva(modtrav)` structura arborelui este definită în felul următor:

```
type tipref = 0.. 2;
       ref = ^varf;
      varf = record
               cod:Integer;
               refs:array[tipref] of ref;
               indice:0..3
             end;
```

Vectorul `refs` va conţine cele trei referinţe: una către nodul tată, celelalte două către fii. Trecerea de la un nod la altul, în traversarea arborelui, este dirijată de valoarea contorului `indice` din nodul curent, care în afară de rolul său menţionat (numără atingerile) va fi şi indice în vectorul de referinţe `refs`. Câmpul `indice` se iniţializează la crearea nodului cu valoarea `3`, ceea ce permite ca actualizarea referinţei la predecesor (care s-ar face greoi la construire) să se facă la prima vizitare a nodului (se verifică uşor că această valoare nu mai poate fi atinsă datorită funcţiei modulo).

```
procedure Traversare_nerecursiva(modtrav:trav);
var p,q:ref;
  gasit:Boolean;
begin
  p:=radacina;                                  { p este nodul curent }
  q:=nil;                                  { iar q este tatăl nodului curent }
  while p <> nil do
    with p^ do begin
      repeat
        if indice = 3 then begin              { prima atingere a nodului }
          refs[0]:=q;                    { se actualizează referinţa la tata }
          q:=p;
          indice:=0
        end;
        if indice = modtrav then
          Write(g,cod:3);                                { prelucrare nod }
        indice:=(indice+1) mod 3;
```

```
        if indice=0 then begin                    { se urcă la tata }
          q:=refs[0];
          gasit:=true
        end else
          gasit:=refs[indice] <> nil
      until gasit;
      p:=refs[indice]
    end;
    WriteLn(g)
end;
```

Procedura `Traversare_nerecursiva(modtrav)` se va apela cu valori diferite ale parametrului `modtrav` şi anume pentru `modtrav = 0` obţinem o traversare în *preordine*, pentru `1` în *inordine*, iar pentru `2` în *postordine*.

Acest algoritm are meritul că este relativ simplu şi rapid, dar are dezavantajul că necesită o memorie suplimentară destul de mare în fiecare nod (referinţa ascendentă şi contorul).

C. Traversarea arborilor oarecare

1. Traversarea pe orizontală

Construirea arborelui se realizează în procedura `Citeste_arbore` prin citirea lui în modul următor: pentru fiecare nod se citeşte perechea (*informaţie utilă*, *număr descendenţi*). Nodurile sunt citite în postordine şi sunt păstrate într-o stivă până când apare nodul ai cărui fii sunt. În acest moment fiii sunt scoşi din stivă şi se actualizează referinţele descendente (de la tată la fii) după care tatăl este pus în stivă; în final singurul nod din stivă va fi rădăcina, iar arborele va fi gata construit.

În Pascal structura se descrie în modul următor:

```
type ref=^nod;
     fiu=0..max;
     nod=record
           info:Integer;
           refs:array[1..max] of ref;
           fii:fiu
         end;
```

unde `max` este o constantă care reprezintă numărul maxim de descendenţi. Procedura `Traversare_pe_orizontala_arbori_oarecare` utilizează o coadă pentru păstrarea nodurilor care urmează să fie prelucrate. Iniţial se introduce rădăcina în coada vidă, după care se scoate pe rând câte un nod din coadă, introducându-se descendenţii lui. Se verifică uşor că acest mecanism asigură într-adevăr parcurgerea arborelui nivel după nivel.

```
procedure Citeste_arbore;                    { citire arbore în postordine }
const max1=11;                          { adâncimea maximă a stivei de noduri }
var n:Integer;                                        { număr total de noduri }
    r:ref;
    stiva:array[1..max1] of ref;
    ist:0..max1;                                      { indicator de stivă }
    fi:fiu;

  procedure Push(p:ref);                          { introducere nod în stivă }
  begin
    if ist = max1 then WriteLn(g,'Stiva plina!')
    else begin
      ist:=ist + 1;
      stiva[ist]:=p
    end
  end;

  function Pop:ref;                                { scoatere nod din stivă }
  begin
    if ist = 0 then
      Pop:=nil                                     { arborele poate fi vid }
    else begin
      Pop:=stiva[ist];
      ist:=ist - 1
    end
  end;

begin                                        { procedura Citeste_arbore }
  ist:=0;                                              { inițializare stivă }
  ReadLn(f,n);                                         { număr total de noduri }
  WriteLn(g,'S-au citit perechile:');
  while n > 0 do begin
    New(r);                                            { creare nod nou }
    with r^ do begin
      ReadLn(f,info,fii);
      for fi:=fii downto 1 do
        refs[fi]:=Pop                    { se scot descendenții de pe stivă }
    end;
    Write(g,'(',r^.info:3,',',r^.fii:3,') ');
    Push(r);                                  { noul nod se pune pe stivă }
    n:=n-1;
    if n mod 4=0 then WriteLn(g)
  end;
  rad:=Pop             { singurul nod pe stivă trebuie să fie rădăcina arborelui }
end;
```

Traversarea în lățime (nivel după nivel) a unui arbore oarecare (pentru a-l afișa) este prezentată în procedura `Traversare_pe_orizontala_arbori_oarecare`.

```
procedure Traversare_pe_orizontala_arbori_oarecare;
const max2=20;                                  { lungimea maximă a cozii }
var r:ref;
    coada:array[0..max2] of ref;
    inda,inde:0..max2;        { indicator de adăugare, respectiv de eliminare }
    fi:fiu;

  procedure Adaug(p:ref);                       { adăugare nod în coadă }
  begin
    if inde = (inda+1) mod max2 then WriteLn(g,'Coada plina!!')
    else begin
      coada[inda]:=p;
      inda:=(inda+1) mod max2
    end
  end;

  function Elim:ref;                            { scoatere nod din coadă }
  begin
    if inda = inde then elim:=nil          { s-a parcurs întregul arbore }
    else begin
      Elim:=coada[inde];
      inde:=(inde+1) mod max2
    end
  end;

begin     { instrucțiunea compusă a procedurii de traversare (vom afișa arborele) }
  inda:=0;
  inde:=0;                                      { inițializare coadă }
  WriteLn(g);
  WriteLn(g,'Arborele tiparit nivel dupa nivel:');
  WriteLn(g);
  Adaug(rad);                                   { se începe cu rădăcina }
  repeat
    r:=elim;                                    { următorul nod }
    if r <> nil then
      with r^ do begin
        Write(g,info:3);
        for fi:=1 to fii do
          Adaug(refs[fi])          { introducerea descendenților în coadă }
      end
  until r=nil;
  WriteLn(g)
end;
```

2. Traversarea în adâncime
Spre deosebire de traversarea în inordine, care are un sens bine determinat doar pentru arbori binari, traversările în preordine şi postordine se pot folosi şi în cazul arborilor oarecare. Dacă arborele oarecare este reprezentat cu referinţe descendente, atunci traversarea se realizează printr-o simplă generalizare a metodei care se aplică la arborii binari.

Dacă arborele este reprezentat prin arborele binar echivalent, atunci se poate arăta că parcurgerea în preordine a arborelui binar echivalent corespunde parcurgerii în preordine a arborelui iniţial, iar parcurgerea în inordine a arborelui binar ataşat corespunde parcurgerii în postordine a arborelui oarecare. Observăm că traversării în postordine a arborelui echivalent nu îi putem găsi un corespondent.

9.6.3. Arbori de căutare

Structurile arborescente sunt foarte des folosite pentru memorarea şi regăsirea rapidă a unor informaţii. Informaţiile memorate pot fi înregistrări oricât de complexe, dar ele conţin un câmp numit *cheie*, care serveşte la identificarea înregistrării. Din punctul de vedere al operaţiilor care se efectuează asupra arborilor de căutare, doar câmpul cheie este important, de celelalte putem face abstracţie.

Fie C mulţimea cheilor posibile ale înregistrărilor care vor trebui regăsite cu ajutorul arborelui de căutare. Dacă arborele de căutare este astfel construit încât foloseşte o relaţie de ordine totală pe C, atunci vom spune că arborele de căutare este *bazat pe ordinea cheilor*. În esenţă, într-un arbore de căutare bazat pe ordinea cheilor căutarea decurge în felul următor: se compară cheia căutată cu cheile din nodurile arborelui, de fiecare dată alegând un subarbore sau altul în funcţie de rezultatul comparării (mai mic, mai mare) până când este găsită înregistrarea căutată (sau se ajunge la concluzia că lipseşte din arbore, caz în care avem căutare fără succes).

În continuare, ne vom ocupa de arborii de căutare bazaţi pe ordinea cheilor.

Fie $S \subseteq C$ mulţimea cheilor care conduc la o căutare cu succes în arbore. Un arbore de căutare bazat pe ordinea cheilor este aşadar un arbore în nodurile căruia vom găsi elemente din C astfel ataşate nodurilor încât să ne putem folosi de relaţia de ordine din C pentru regăsirea elementelor din S. Există două abordări posibile:
1. În *abordarea cu informaţia în frunze* (*leaf-oriented*) frunzele corespund înregistrărilor memorate în arbore, iar în nodurile neterminale apar elemente din C care au rolul de a dirija căutarea care se termină întotdeauna la o frunză. În această abordare este preferabil ca arborele să fie neomogen, adică frunzele să corespundă înregistrărilor memorate în arbore (identificate prin câmpul cheie), iar nodurile neterminale să fie înregistrări care conţin referinţele către celelalte noduri din arbore cu care sunt în legătură şi o cheie din C (sau mai multe la arborii *multicăi*).

2. În *abordarea cu informația în noduri* (*node-oriented*) atât în nodurile terminale cât şi în cele neterminale vom avea înregistrări memorate în arbore (această structură este omogenă).

Arborii de căutare, bazați pe ordinea cheilor, pot fi de două feluri: *binari* sau *multicăi*. Arborii binari de căutare au câte o singură cheie asociată fiecărui nod, arborii multicăi au mai multe.

În continuare ne vom ocupa de arborii binari de căutare.

Într-un arbore binar de căutare cheile sunt astfel ataşate nodurilor încât traversând arborele în inordine, cheile vor fi întâlnite în ordine crescătoare (strict crescătoare în cazul abordării cu informația în noduri).

În continuare, arborii binari de căutare *se vor trata în abordarea cu informația în noduri*. Se ştie că dacă mulțimea S are n elemente, atunci pentru această mulțime se poate construi un arbore binar de căutare total echilibrat de înălțime $[\log_2 n]$. (Înălțimea unui arbore reprezintă numărul de niveluri în arborele respectiv.) Rezultă că se vor face cel mult $[\log_2 n]$ comparații de chei pentru a găsi nodul cu cheia căutată sau pentru a constata că elementul lipseşte.

Amintim că şi în cazul abordării cu informația în frunze înălțimea arborelui total echilibrat este tot $O(\log_2 n)$ deoarece se verifică uşor că un arbore binar cu n frunze are $n - 1$ noduri neterminale.

Algoritmul de căutare într-un arbore binar de căutare poate fi descris recursiv în felul următor:

1. dacă arborele este vid avem *căutare fără succes*, deci algoritmul se termină;
2. se compară cheia x cu cheia c a rădăcinii; dacă $x = c$ avem *căutare cu succes* (algoritmul se termină);
3. dacă $x < c$, atunci se continuă căutarea în subarborele stâng;
4. altfel $x > c$ şi continuăm căutarea în subarborele drept.

A. Inserarea în arbori de căutare

În practică există foarte multe aplicații în care căutarea se combină cu inserarea. În momentul în care căutarea se termină fără succes, elementul căutat se introduce într-un nod nou, care se leagă în arbore de locul unde ar fi trebuit să se găsească (unde a eşuat căutarea).

Se observă că inserarea nu se poate face eficient în cazul reprezentării liniare, deoarece ar necesita deplasarea unor elemente, în schimb, în cazul unei reprezentări semistatice sau dinamice această operație este foarte simplă.

Algoritmul următor – dat fiind faptul că pleacă de la arborele vid în care inserează primul element căutat (pe care evident nu-l găseşte) ş.a.m.d. – este de fapt un algoritm de construire pentru arborele de căutare:

1. dacă arborele este vid, se creează un nou nod care va fi nodul rădăcină; cheia va avea valoarea căutată *x*;
2. dacă cheia *c* a rădăcinii este egală cu *x*, algoritmul se termină;
3. dacă $x < c$, se reia algoritmul pentru subarborele stâng;
4. altfel $x > c$; se reia algoritmul pentru subarborele drept.

Căutarea unei chei *x* într-un arbore total echilibrat cu *n* noduri necesită cel mult $N_c = [\log_2 n]$ comparații. În general, un arbore de căutare nu este total echilibrat, deci în procedura `Cauta(x,p)` numărul comparațiilor necesare în căutare va fi mai mare decât N_c. Cazul cel mai defavorabil este acela când toate cheile care urmează să fie inserate sosesc în ordine crescătoare sau descrescătoare, deoarece în acest caz arborele degenerează într-o listă liniară în care numărul de comparații necesare pentru o căutare este în medie *n*/2. Prin urmare, deşi revenirea la forma de arbore total echilibrat după inserare ar reduce lungimea medie a căutării, totuşi transformările necesare fiind laborioase, ele nu se justifică în general. De aceea, în practică se folosesc concepte mai puțin restrictive de echilibru, ceea ce permite obținerea unor algoritmi eficienți.

```
procedure Cauta(x:Integer; var p:ref);
                  { x este cheia nouă; se va returna p, pointer la nodul nou inserat }
begin                              { construirea arborelui de căutare prin inserare }
  if p = nil then begin                                        { căutare fără succes }
    New(p);
    with p^ do begin
      cod:=x;
      stang:=nil;
      drept:=nil
    end
  end else
    if x < p^.cod then
      Cauta(x,p^.stang)
    else
      if x > p^.cod then
        Cauta(x,p^.drept)
      else                         { există nod având cheia x, nu îl mai introducem }
end;
```

B. Ştergerea în arbori de căutare

Problema ştergerii se rezolvă simplu dacă dorim să ştergem o cheie care este ataşată unui vârf terminal sau unui vârf cu un singur descendent. Dacă vârful respectiv are doi descendenți, elementul şters va fi înlocuit ori cu nodul cel mai din dreapta al subarborelui stâng, ori cu nodul cel mai din stânga al subarborelui drept, astfel păstrându-se relația de ordine între nodurile arborelui.

Procedura `Sterge(x,p)` face distincție, între următoarele trei cazuri:
1) cheia *x* nu se găseşte în arbore;
2) nodul cu cheia *x* are cel mult un descendent;
3) nodul cu cheia *x* are doi descendenți.

```
procedure Sterge(x:Integer; var p:ref);
var q:ref;                    { ştergerea unui element din arborele de căutare }

  procedure Ster(var r:ref);      { se şterge fizic cel mai din dreapta nod al }
  begin                     { subarborelui stâng al nodului care se şterge logic }
    if r^.drept <> nil then
      Ster(r^.drept)
    else begin
      q^.cod:=r^.cod;
      q:=r;
      r:=r^.stang;
      Dispose(q)
    end
  end;

begin                                                  { procedura Sterge }
  if p = nil then
    Write(g,'Nodul ',x,' nu exista, deci nu poate fi sters).')
  else
    if x < p^.cod then
      Sterge(x,p^.stang)
    else
      if x > p^.cod then
        Sterge(x,p^.drept)
      else begin
        q:=p;
        if q^.drept = nil then begin            { nodul nu are fiu drept }
          p:=q^.stang;
          Dispose(q)
        end else
          if q^.stang = nil then begin          { nodul nu are fiu stâng }
            p:=q^.drept;
            Dispose(q)
          end else
            Ster(q^.stang)                          { nodul are doi fii }
      end
end;
```

9.6.4. Sortarea cu arbori binari

Considerăm o mulțime de articole pe care dorim să le sortăm crescător după valorile unui anumit câmp.

Există un număr foarte mare de metode de sortare; vom aminti doar două dintre ele, care realizează şiruri ordonate de elemente utilizând o structură arborescentă.

Prima metodă, *metoda sortării prin inserare binară*, constă în a insera elementul a_k, $k \in \{2, ..., n\}$ în şirul ordonat $a_1 \le a_2 \le ... \le a_{k-1}$, determinându-i locul prin căutare binară. Mai general, sortarea se poate realiza inserând pe rând elementele într-un arbore de căutare bazat pe ordinea cheilor, inițial vid şi traversând după aceea arborele în inordine. Subprogramul `Cauta(x,p)` poate fi considerat o procedură de sortare prin inserare.

Fie $a_1, a_2, ..., a_n$, un vector cu n componente. Alegând arbitrar un arbore binar cu n vârfuri, putem ataşa vârfurilor sale câte o componentă a vectorului dat, astfel încât pentru orice vârf i, cheia ataşată lui să fie mai mică sau egală cu cheile ataşate descendenților săi. Odată realizat acest lucru, cheia ataşată rădăcinii are valoarea cea mai mică dintre elementele vectorului. Bazându-ne pe acest lucru, vom descrie următorul algoritm de sortare: memorăm valoarea ataşată rădăcinii şi printr-un sistem de „promovare" în ierarhia determinată în arbore, obținem un arbore cu $n - 1$ vârfuri care păstrează proprietățile enunțate anterior, proprietăți ce caracterizează o grupare denumită *ansamblu* (*heap*). Repetând procedeul de n ori, obținem elementele vectorului în ordine crescătoare.

Numim *ansamblu* un arbore binar în care pentru orice vârf i valoarea a_i ataşată lui este mai mică sau egală cu oricare dintre valorile ataşate descendenților săi.

Un ansamblu poate fi reprezentat foarte eficient în memorie cu ajutorul unei reprezentări liniare într-un tablou cu n elemente. Rădăcina arborelui este primul element din tablou, adică elementul cu indicele 1. Fiul drept al elementului cu indicele i este elementul cu indicele $2i + 1$, iar fiul stâng este cel cu indicele $2i$, pentru orice $i \ge 1$ pentru care $2i + 1 \le n$, respectiv $2i \le n$. Se poate verifica foarte uşor că fiecărui element al tabloului îi corespunde un nod în arbore, deci numărul nodurilor din arbore va fi n. Aşadar, structura arborescentă apare numai implicit şi numai vectorul (a_i), $i = 1, ..., n$ are nevoie de spațiu în memorie. Putem spune că un vector (a_i), $i = 1, ..., n$ formează un ansamblu dacă:

- pentru orice $i \in \{1, 2, ..., [n/2]\}$ avem $a_i \le a_{2i}$ şi
- pentru orice i, $1 < 2i + 1 \le n$ avem $a_i \le a_{2i+1}$.

Introducerea noțiunii de ansamblu determină necesitatea soluționării a două probleme importante:

1) fiind dat un vector cu n componente, să se formeze din el un ansamblu.
2) fiind dat un ansamblu $a_1, a_2, ..., a_i$, să se schimbe a_1 cu a_i şi să se reconstituie din primele $i - 1$ componente ale vectorului (a_i), $i = 1, ..., n$ un ansamblu.

```
procedure Sort(n:Byte; var x:sir);
var i:Byte;

  procedure Sch(i,j:Byte);              { interschimbarea a două elemente }
  var a:Integer;
  begin
    a:=x[i];
    x[i]:=x[j];
    x[j]:=a
  end;

  procedure Coborare(i,n:Byte);
                          { al i-lea element adăugat poate coborî în structură, }
  var j:Byte;                  { astfel încât şirul x să formeze un ansamblu }
      stop:Boolean;
  begin
    stop:=false;                                        { la apel n ≥ 2i }
    while not stop do begin
      j:=2*i;                                           { descendent stâng }
      if j < n then                                { există şi descendent drept }
        if x[j+1] > x[j] then
          j:=j+1;                                { j este indicele elementului maxim }
      if x[i] >= x[j] then
        stop:=true                                   { nu mai avem ce modifica }
      else begin
        sch(i,j);                                 { elementul i coboară în locul lui j }
        i:=j;
        stop:=2*i>n        { coborârea poate continua dacă sunt descendenți }
      end
    end
  end;

  begin                                                 { procedura Sort }
    for i:=n div 2 downto 2 do
      Coborare(i,n);           { elementele se adaugă în vârf unul câte unul }
    for i:=n downto 2 do begin
      Coborare(1,i);
      sch(1,i) { scoaterea elementului maxim şi adăugarea unui element în vârf }
    end
  end;
```

9.6.5. Arbori echilibraţi AVL

Având în vedere că algoritmul de inserare într-un arbore binar de căutare reprezentat cu referinţe explicite este foarte complicat în cazul în care se doreşte să se păstreze structura de arbore total echilibrat, *Adelson-Velskii* şi *Landis* au propus utilizarea unei noţiuni mai puţin restrictive, cea de *arbore echilibrat după înălţime* la care lungimea medie de căutare nu diferă în mod substanţial faţă de arborele total echilibrat.

Un arbore binar se va numi *echilibrat după înălţime* dacă *diferenţa de înălţime dintre cei doi subarbori ai oricărui vârf este cel mult* 1. Aceşti arbori se mai numesc şi *arbori AVL* după numele celor care i-au definit. Din definiţie reiese clar că orice subarbore al unui arbore AVL este tot un arbore AVL.

Observăm că orice arbore total echilibrat este şi echilibrat. Rezultă că arborii total echilibraţi sunt un caz particular al celor echilibraţi.

Un arbore binar AVL va fi cu cel mult 45% mai înalt decât unul total echilibrat cu acelaşi număr de noduri (afirmaţie demonstrată de Adelson-Velskii şi Landis). Deci lungimea drumului de căutare nu creşte considerabil. Numărul de comparaţii realizate prin operaţiile de căutare, inserare şi ştergere va fi tot de ordinul $O(\log_2 n)$.

A. Inserarea într-un arbore AVL

Să presupunem că trebuie să inserăm un nod în subarborele stâng al unui arbore AVL şi că în urma acestei inserări înălţimea subarborelui stâng ar creşte. Dacă notăm cu $i(st)$ respectiv $i(dr)$ înălţimea celor doi subarbori (stâng respectiv drept) înainte de inserare, putem distinge următoarele trei cazuri:

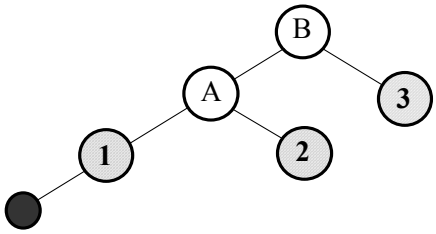
1) dacă $i(st) = i(dr) \Rightarrow$ prin inserare se va strica egalitatea dintre înălţimile subarborilor, dar proprietatea de echilibru se menţine;
2) dacă $i(st) < i(dr) \Rightarrow$ după inserare cei doi subarbori vor avea aceeaşi înălţime, deci caracteristica de arbore echilibrat se păstrează;
3) dacă $i(st) > i(dr) \Rightarrow$ prin inserare se strică proprietatea de echilibru, trebuie reechilibrat arborele. Se poate demonstra că problema reechilibrării se reduce la cele două cazuri şi simetricele lor pentru inserarea în subarborele drept, prezentate mai jos.

Fie • nodul nou inserat care şi-a găsit loc în subarborele stâng al arborelui:

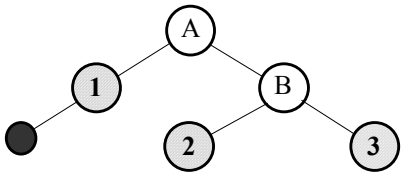
Cazul I.

Citind cheile în inordine obţinem:

•, 1, A, 2, B, 3

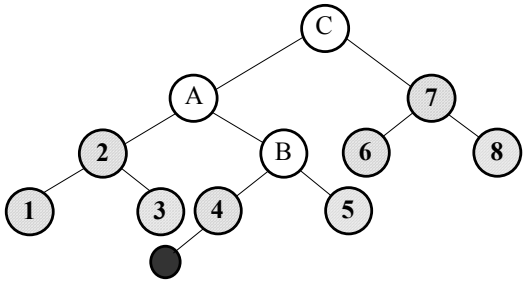


Caracteristica de arbore echilibrat se poate restabili în felul următor:

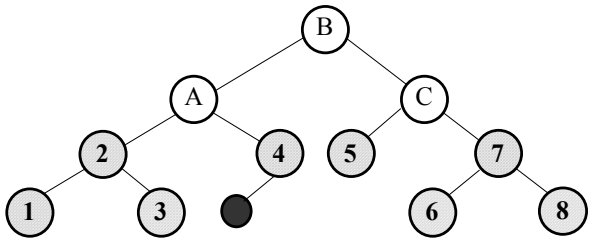


Şirul cheilor în inordine rămâne neschimbat:
•, 1, A, 2, B, 3

Cazul II.
Şirul cheilor în inordine:
1, 2, 3, A, •, 4, B, 5, C, 6, 7, 8



În acest caz rotaţiile pentru restabilirea proprietăţii de echilibru duc la următoarea structură:



Şirul cheilor rămâne acelaşi:
1, 2, 3, A, •, 4, B, 5, C, 6, 7, 8

Se observă că vârfurile se mută de pe un nivel pe altul dintr-un subarbore în altul, dar traversarea în inordine realizează aceeaşi rută prin vârfuri, (deci relaţia de ordine totală definită de arborele de căutare se păstrează).

Vom introduce noţiunea de *factor de echilibru* al unui nod, care reprezintă diferenţa dintre înălţimea subarborelui stâng şi al celui drept, deoarece algoritmul de inserare în arbori AVL depinde în mare măsură de valoarea acestui factor în nodurile care se află între rădăcină şi locul unde trebuie inserat noul nod.

Astfel algoritmul se va desfăşura în trei faze:
1) căutarea elementului până când se constată absenţa lui;
2) inserarea noului nod şi stabilirea factorului de echilibru;
3) verificarea factorilor de echilibru pe drumul căutării (parcurs de jos în sus) şi efectuarea eventualei reechilibrări.

Cele trei situaţii referitoare la înălţimile subarborilor afectaţi corespund valorii factorului de echilibru respectiv egal cu 0, -1, +1. În cea de-a treia situaţie este necesară şi ramificarea algoritmului în funcţie de tipurile de dezechilibrare (tip I sau II) tratate anterior. Operaţia de restabilire a echilibrului se realizează printr-o permutare circulară a referinţelor, permutare care implică două vârfuri în cazul I (rotaţie simplă) sau trei vârfuri în cazul II (rotaţie dublă). Această modificare a referinţelor determină şi modificarea factorului de echilibru în fiecare nod implicat în rotaţie.

Structura arborescentă o vom descrie într-un mod asemănător celui din programele precedente, adăugându-se informaţia referitoare la factorul de echilibru, utilizată în algoritm pentru reechilibrare, dacă este cazul. Avem aşadar:

```
const stanga=-1;
      echilibru=0;
      dreapta=1;
      st=0;
      dr=1;

type dir=st..dr;
     ech=stanga..dreapta;
     ref=^varf;
     varf=record
            cod:Integer;
            fin:array[dir] of ref;
            fact:ech
          end;

function Cauta(x:Integer):ref;          { construirea arborelui echilibrat }
var m:Boolean;
    rez:ref;

  procedure Reechil(var p:ref; d:ech);
  var stang,drept:dir;
      p1,p2:ref;
      e:ech;
  begin
    if m then begin                     { s-a modificat factorul de echilibru }
      if d = stanga then begin
        stang:=st;
        drept:=dr
      end else begin
        stang:=dr;
        drept:=st
      end;
```

```
      e:=p^.fact;
      if e = echilibru then
        p^.fact:=d                          { modificarea se propagă mai sus }
      else begin
        m:=false;                           { modificarea nu se propagă }
        if e*d = stanga then
          p^.fact:=echilibru
        else begin                                          { reechilibrare }
          p1:=p^.fiu[stang];
          if p1^.fact = d then begin                        { rotație simplă }
            p^.fiu[stang]:=p1^.fiu[drept];
            p1^.fiu[drept]:=p;
            p^.fact:=echilibru;
            p:=p1
          end else begin                                    { rotație dublă }
            p2:=p1^.fiu[drept];
            p1^.fiu[drept]:=p2^.fiu[stang];
            p2^.fiu[stang]:=p1;
            p^.fiu[stang]:=p2^.fiu[drept];
            p2^.fiu[drept]:=p;
            if p2^.fact = d then
              p^.fact:=-d
            else
              p^.fact:=echilibru;
            if p2^.fact = -d then
              p1^.fact:=d
            else
              p1^.fact:=echilibru;
            p:=p2
          end;
          p^.fact:=echilibru
        end
      end
    end
  end;                                          { sfârşit procedura Reechil }

  procedure Caut(var p:ref);
  begin
    if p = nil then begin                   { elementul căutat nu a fost găsit }
      New(p);                                           { se creează noul nod }
      m:=true;                  { se atenționează nivelul superior că eventual }
                                                  { s-a produs o dezechilibrare }
      with p^ do begin
        cod:=x;
```

```
        fiu[st]:=nil;                          { noul nod este frunză }
        fiu[dr]:=nil;
        fact:=echilibru
      end;
      rez:=p
    end else
      if x < p^.cod then begin
        Caut(p^.fiu[st]);      { căutarea se continuă în subarborele stâng }
        Reechil(p,stanga)
      end else
        if x > p^.cod then begin
          Caut(p^.fiu[dr]);    { căutarea se continuă în subarborele drept }
          Reechil(p,dreapta)
        end else begin
          m:=false;                            { cheia a fost găsită }
          rez:=p
        end
  end;                                         { sfârşit procedura Caut }

begin                                          { procedura Cauta }
  Caut(radacina);
  Cauta:=rez
end;
```

B. Ştergerea în arbori echilibraţi AVL

Arborele poate să se dezechilibreze şi în urma unei ştergeri. Echilibrul pierdut poate fi restabilit şi în cazul acesta prin rotaţia referinţelor.

Din punct de vedere al numărului de rotaţii necesare pentru reechilibrare, diferenţa esenţială dintre inserare şi ştergere constă în faptul că inserarea necesită cel mult o rotaţie care implică două sau trei vârfuri, iar ştergerea poate necesita rotaţii pentru nodurile de pe întreg drumul de căutare. Este de remarcat că după aproximativ fiecare a doua inserare şi după aproximativ fiecare a cincea ştergere trebuie aplicate rotaţii de reechilibrare.

```
procedure Sterge(x:Integer);
var m:Boolean;
    q:ref;

  procedure Reech_s(var p:ref; d:ech);
                  { modificarea factorului de echilibru şi eventual de reechilibru }
  var stang,drept:dir;
      p1,p2:ref;
      e1,e2:ech;
```

```
begin
  if m then begin
    e1:=p^.fact;
    if e1 = d then
      p^.fact:=echilibru              { modificarea se propagă mai sus }
    else
      if e1 = echilibru then begin      { modificarea nu se propagă }
        p^.fact:=-d;
        m:=false
      end else begin
        if d = stanga then begin
          stang:=st;
          drept:=dr
        end else begin
          stang:=dr;
          drept:=st
        end;
        p1:=p^.fiu[drept];
        e1:=p1^.fact;
        if e1 <> d then begin              { rotație simplă dreapta }
          p^.fiu[drept]:=p1^.fiu[stang];
          p1^.fiu[stang]:=p;
          if e1 = echilibru then begin  { nu se propagă modificarea }
            p^.fact:=-d;
            p1^.fact:=d;
            m:=false
          end else begin                  { modificarea se propagă }
            p^.fact:=echilibru;
            p1^.fact:=echilibru
          end;
          p:=p1
        end else begin                                { rotație dublă }
          p2:=p1^.fiu[stang];
          p1^.fiu[stang]:=p2^.fiu[drept];
          p2^.fiu[drept]:=p1;
          p^.fiu[drept]:=p2^.fiu[stang];
          p2^.fiu[stang]:=p;
          e2:=p2^.fact;
          if e2 = -d then p^.fact:=d
                     else p^.fact:=echilibru;
          if e2 = d then p1^.fact:=-d
                    else p1^.fact:=echilibru;
          p:=p2;
          p2^.fact:=echilibru
        end
      end
  end
end;                                  { sfârşit procedura Reech_s }
```

```
  procedure Ster(var r:ref);        { eliminarea celui mai din dreapta nod }
  begin          { al subarborelui stâng, cu reechilibrare pe calea de revenire }
    if r^.fiu[dr] <> nil then begin
      Ster(r^.fiu[dr]);                                  { avansare spre dreapta }
      Reech_s(r,dreapta)                  { se modifică factorul de echilibru }
    end else begin                  { suntem în nodul care se va şterge fizic }
      q^.cod:=r^.cod;    { conţinutul lui se pune în nodul care se şterge logic }
      q:=r;
      r:=r^.fiu[st];
      Dispose(q);
      m:=true
    end
  end;                                          { sfârşit procedura Ster }

  procedure Caut_s(var p:ref);
  begin
    if p = nil then begin
      Write(g,' Codul nu este in arbore!');
      m:=false
    end else
      if x < p^.cod then begin
        Caut_s(p^.fiu[st]);                { căutare în subarborele stâng }
        Reech_s(p,stanga)             { modificarea factorului de echilibru }
      end else
        if x > p^.cod then begin
          Caut_s(p^.fiu[dr]);              { căutare în subarborele drept }
          Reech_s(p,dreapta)
        end else begin             { suntem în nodul care trebuie şters logic }
          q:=p;
          if q^.fiu[dr] = nil then begin      { nu are subarbore drept }
            p:=q^.fiu[st];
            Dispose(q);
            m:=true
          end else
            if q^.fiu[st] = nil then begin    { nu are subarbore stâng }
              p:=q^.fiu[dr];
              Dispose(q);
              m:=true
            end else begin                                { doi descendenţi }
              Ster(q^.fiu[st]);
              Reech_s(p,stanga)
            end
          end
  end;                                        { sfârşit procedura Caut_s }
begin                                                  { procedura Sterge }
  Caut_s(radacina)
end;
```

9.7. Implementări sugerate

Modul de lucru în condițiile alocării dinamice diferă esențial de modul în care s-a lucrat în alocarea statică. Recomandăm, pentru a forma deprinderile necesare rezolvării de probleme, să implementați următoarele structuri de date, astfel încât să exersați realizarea operațiilor specifice acestora. Ar fi utilă crearea unui unit pentru prelucrarea structurilor de tip:

1. stivă;
2. coadă;
3. coadă cu santinelă;
4. listă simplu înlănțuită;
5. listă ordonată;
6. listă dublu înlănțuită;
7. listă dublu înlănțuită cu santinele;
8. listă circulară;
9. arbore binar total echilibrat;
10. arbore oarecare;
11. arbore binar de căutare;
12. arbore AVL.

9.8. Probleme propuse

9.8.1. Polinom

Fie două polinoame P(X) şi Q(X). Afişați polinomul S(X) = P(X) + Q(X).

Date de intrare

Fişierul **`POLINOM.IN`** conține date referitoare la monoamele celor două polinoame. Pe prima linie a fişierului se află un număr natural m, reprezentând numărul monoamelor primului polinom. Pe următoarele m linii sunt descrise cele m monoame. Un monom este caracterizat prin coeficientul său şi gradul necunoscutei. Cele două numere sunt despărțite printr-un spațiu. Pe următoarea linie se află un număr natural n, reprezentând numărul monoamelor celui de al doilea polinom. Pe următoarele n linii sunt descrise cele n monoame în mod similar primului polinom.

Date de ieşire

Fişierul de ieşire **`POLINOM.OUT`** va conține datele referitoare la monoamele polinomului sumă. Fiecare linie a fişierului va conține descrierea unui monom în acelaşi format ca în fişierul de intrare.

Restricții și precizări
- Coeficienții sunt numere întregi (-32000 ≤ *coef* ≤ 32000);
- Gradele sunt numere naturale (0 ≤ *grad* ≤ 1000);
- Ordinea monoamelor este descrescătoare după gradul lui *X*.

Exemplu

POLINOM.IN
```
3
-2 2
1 1
-3 0
2
2 2
1 1
```

POLINOM.OUT
```
2 1
-3 0
```

9.8.2. Farfurii

Modelați următoarea activitate:
a) Se despachetează o ladă în care au fost ambalate mai multe farfurii și se așează în vraf. Farfuriile sunt caracterizate de modelul cu care au fost ornate; modelele sunt distincte.
b) Se caută o farfurie cu un anumit model dat. Farfuriile verificate se așează într-un vraf nou până când se găsește farfuria căutată sau se constată că ea nu există printre farfuriile despachetate (deci s-a spart).

Date de intrare
În fișierul de intrare **VRAF.IN** se află un număr necunoscut de șiruri de caractere, câte un șir pe o linie, reprezentând modele de farfurii. Primul model este cel căutat, restul sunt modelele de pe farfuriile scoase din ladă care se vor pune în vraf.

Date de ieșire
Pe prima linie a fișierului de ieșire **VRAF.OUT** se va scrie cuvântul `'DA'` sau `'NU'`, în funcție de rezultatul căutării farfuriei cu modelul dat (dacă s-a găsit farfuria se scrie `'DA'`). Pe a doua linie se va scrie un număr natural *k*, reprezentând numărul farfuriilor verificate și mutate în noul vraf în timpul căutării farfuriei având modelul dat. Pe următoarele *k* linii se vor scrie modelele farfuriilor mutate în noul vraf în timpul căutării. Ordinea lor de afișare este de la vârf spre bază în noul vraf.

Restricții și precizări
- un model este format din cel mult 20 de caractere litere mici ale alfabetului englezesc;
- în ladă sunt cel mul 1000 de farfurii.

Exemplu

VRAF.IN
```
maci
viorele
trandafiri
maci
ghiocei
fructe
```

VRAF.OUT
```
DA
2
fructe
ghiocei
```

9.8.3. Agenție

O agenție organizează o excursie în Hawai cu prețuri promoționale la care doresc să participe *n* persoane. Agenția a decis ca în momentul în care află numărul de locuri *k*, să trimită în excursie primele *k* persoane în ordinea înscrierii. Afișați participanții la excursie.

Date de intrare

Pe prima linie a fișierului de intrare **HAWAI.IN** se află un număr natural *n*, reprezentând numărul persoanelor înscrise la agenție. Pe următoarele *n* linii se află câte un șir de caractere, reprezentând numele persoanelor înscrise la excursie. Pe următoarea linie se află un număr natural *k*, reprezentând numărul de locuri.

Date de ieşire

Fişierul de ieşire **HAWAI.OUT** va conține numele persoanelor care vor participa la excursie, în ordinea înscrierii.

Restricții şi precizări

- $1 \le n, k \le 1000$;
- numele persoanelor sunt formate din cel mult 20 de caractere, litere mici ale alfabetului englezesc.

Exemplu

HAWAI.IN
```
7
Popescu Nicolae
Bara Mircea
Bara Maria
Lungu Vasile
Petrescu Ana
Mircescu Adrian
Ionescu Ilie
5
```

HAWAI.OUT
```
Popescu Nicolae
Bara Mircea
Bara Maria
Lungu Vasile
Petrescu Ana
```

9.8.4. Mesaje

Vasile vrea să trimită mesaje de felicitare prietenilor săi pe Internet. Deoarece numărul *n* de adrese pe care le are este mare, decide să trimită mesaje doar la fiecare a *k*-a adresă din lista de adrese. Atunci când ajunge la sfârşitul listei, continuă numărătoarea de la început, ca şi cum adresele ar fi aşezate în cerc. Se opreşte în momentul în care ar trebui să trimită un mesaj la cineva a doua oară. Afişaţi numele prietenilor lui Vasile care au primit mesaj de felicitare.

Date de intrare

Pe prima linie a fişierului de intrare **INTERNET.IN** se află două numere naturale *n* şi *k*, având semnificaţiile din enunţ. Pe fiecare din următoarele *n* linii se află un şir de caractere, reprezentând numele unui prieten.

Date de ieşire

În fişierul de ieşire **INTERNET.OUT** se vor scrie numele prietenilor lui Vasile care au primit mesaj de felicitare.

Restricţii şi precizări

- $1 \le n, k \le 10000$;
- În timpul numărătorii se numără şi persoanele care au primit deja mesaj;
- numele sunt formate din cel mult 20 de litere mici ale alfabetului englezesc.

Exemplu

INTERNET.IN
```
6 4
petru
ana
mircea
maria
mihai
cristina
```

INTERNET.OUT
```
maria
ana
cristina
```

9.8.5. Admitere

La admiterea de la Facultatea de Informatică s-au înscris *n* candidaţi pe *k* locuri existente în anul I. După examen s-au întocmit listele reuşiţilor în funcţie de medii, dar listele devin definitive doar după ce candidaţii îşi depun diploma de bacalaureat în original la secretariatul facultăţii. Cunoscând numele celor care s-au înscris precum şi mediile obţinute de ei, afişaţi lista corespunzătoare situaţiei imediat după examen, apoi, pe baza numelor celor care şi-au prezentat diploma de bacalaureat în original, afişaţi lista studenţilor de anul I.

Date de intrare
Pe prima linie a fişierului de intrare **ADMITERE.IN** se află două numere naturale n şi k, având semnificaţia din enunţ. Pe următoarele n perechi linii se află numele candidaţilor înscrişi şi mediile lor (vezi exemplu). Pe următoarele linii se află numele celor care au prezentat diploma de bacalaureat.

Date de ieşire
În fişierul de ieşire se vor scrie numele studenţilor înscrişi în anul I, câte un nume pe o linie, în ordinea mediilor.

Restricţii şi precizări
- $1 \le n, k \le 1000$;
- numele sunt formate din cel mult 20 de caractere, litere mici ale alfabetului englezesc;
- mediile sunt numere reale, cu două zecimale exacte;
- chiar dacă mai sunt locuri libere, pentru a fi înscris, candidatul trebuie să aibă media egală cu cel puţin 5;
- în caz de medii egale, numele se vor afişa în ordine alfabetică.

Exemplu

ADMITERE.IN
```
7 5
Popescu Nicolae
7.50
Bara Mircea
8.25
Bara Maria
4.00
Lungu Vasile
3.70
Petrescu Ana
10.00
Mircescu Adrian
9.75
Ionescu Ilie
5.50
Popescu Nicolae
Mircescu Adrian
Bara Mircea
Bara Maria
Lungu Vasile
Ionescu Ilie
```

ADMITERE.OUT
```
Mircescu Adrian
Bara Mircea
Popescu Nicolae
Ionescu Ilie
```

9.8.6. Pregătirea mesei

Bubulina doreşte să facă o surpriză părinţilor ei şi să pregătească masa de prânz. S-a înarmat cu o carte de bucate, alimente şi condimente, un şorţ mare şi e nedumerită de ordinea în care trebuie să realizeze operaţiile pentru a pregăti o masă bună.

Ştie că anumite operaţii se realizează înaintea altora, de exemplu cartofii se spală înainte de a-i fierbe şi doar după ce au fiert se prepară.

Cunoscând numărul operaţiilor pe care trebuie să le realizeze Bubulina şi priorităţile unor operaţii faţă de altele, realizaţi o ordonare liniară a operaţiilor astfel încât masa să fie pregătită foarte bine.

Date de intrare

Pe prima linie a fişierului de intrare **PAPA.IN** se găseşte un număr n care reprezintă numărul de operaţii pe care Bubulina trebuie să le realizeze. Pe următoarele linii se găsesc câte două numere x şi y, separate printr-un spaţiu, reprezentând faptul că operaţia x trebuie realizată înaintea operaţiei y.

Date de ieşire

Fişierul de ieşire **PAPA.OUT** va conţine pe o succesiune de n numere, separate prin câte un spaţiu, reprezentând numerele de ordine ale operaţiilor culinare în ordinea în care pot fi realizate, astfel încât masa să fie bine pregătită.

Restricţii şi precizări

- $1 \le n \le 100$;
- Datele de intrare sunt corecte şi nu există operaţii care să determine o precedenţă circulară – de exemplu operaţia 1 să fie realizată înaintea operaţiei 2, 2 înaintea lui 3 şi 3 înaintea lui 1.

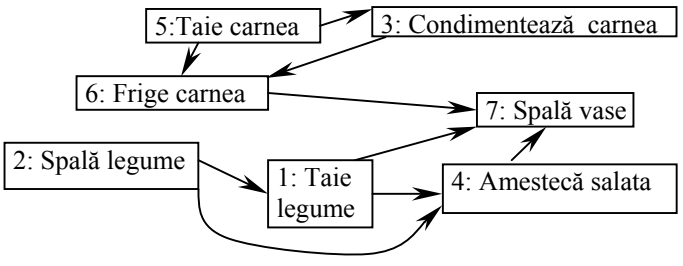
Exemplu

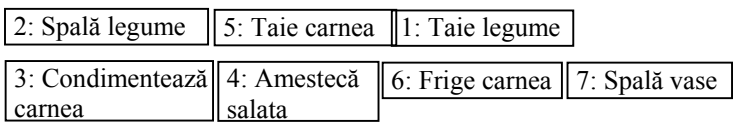
PAPA.IN

```
7
2 1
2 4
1 7
4 7
6 7
5 6
3 6
5 3
1 4
```

PAPA.OUT

```
2 5 1 3 4 6 7
```





9.8.7. Sportivi

Mai mulți sportivi de performanță trebuie să se antreneze pe aparate identice într-o sală de sport. În funcție de nivelul de pregătire, fiecare sportiv se antrenează un anumit timp t_i, stabilit dinainte. Un sportiv se antrenează fără întrerupere şi fără să își schimbe aparatul la care începe antrenamentul. După ce fiecare sportiv îşi termină antrenamentul, şi numai atunci, toți sportivii au voie să meargă, împreună, la „Festivalul Toamnei", care tocmai a început în oraş. Ştiind că sunt m sportivi şi n aparate, repartizați sportivii la aparate, astfel încât să poată pleca la festival cât mai repede.

Date de intrare

Prima linie a fişierului de intrare **SPORTIVI.IN** conține două numere naturale n *şi* m, reprezentând numărul aparatelor, respectiv numărul sportivilor. Următoarea linie conține m numere întregi, reprezentând durata antrenamentului fiecărui sportiv. Aceste numere vor fi separate prin câte un spațiu.

Date de ieşire

Fişierul de ieşire **SPORTIVI.OUT** va conține $n + 1$ linii. Pe prima linie se va scrie un număr întreg care reprezintă timpul scurs de la începerea antrenamentelor până când termină şi ultimul exercițiile şi sportivii pot pleca la festival. Următoarele n linii vor conține duratele antrenamentelor pe aparate, adică pe linia $i + 1$ se vor scrie duratele antrenamentelor sportivilor cărora li s-a repartizat aparatul i.

Restricții şi precizări

- $1 \le n, m \le 1000$;
- $1 \le t_i \le 1000$, unde t_i reprezintă duratele antrenamentelor, $i = 1, 2, \ldots, m$.

Exemplu

SPORTIV.IN	**SPORTIV.OUT**
3 6	9
6 5 1 3 4 7	1 7
	3 6
	4 5

9.8.8. Cutii

S-a mutat muzeul. Obiectele au fost împachetate în cutii având forme cubice de diverse dimensiuni. La despachetare lucrează multe persoane în paralel şi pentru a evita dezordinea, prin încăperile unde se lucrează la despachetare s-a instalat o bandă rulantă pe care se aşează cutiile goale, cu singura deschizătura orientată în sus. Ştefan stă la capătul benzii şi strânge cutiile. El a primit sarcina să împacheteze cutiile unele în altele astfel încât numărul pachetelor de cutii să fie cel mai mic posibil. Directorul muzeului, văzându-l pe Ştefan încurcat datorită acestei cerințe, stabileşte regulile:

- Cutiile se culeg de pe bandă în ordinea sosirii lor.
- Cutia curentă se aşează într-o altă cutie, dacă aceasta are dimensiunea mai mică.
- Dacă nu există pachet început în care să încapă cutia curentă, aceasta va constitui prima cutie dintr-un pachet nou.
- Într-un pachet început se aşează o singură cutie (nu se pun mai multe cutii unele lângă altele într-un pachet început, chiar dacă există loc).
- O cutie aşezată la un moment dat nu se mai scoate.
- Un pachet început nu se aşează în alt pachet, chiar dacă acest lucru ar fi posibil.
- Nici o cutie nu poate fi ignorată.

Scrieți un program care determină numărul minim de pachete de cutii rezultate în urma muncii lui Ştefan, precum şi secvențele de cutii din fiecare pachet.

Date de intrare
Prima linie a fişierului de intrare **CUTII.IN** conține un singur număr natural *n*, reprezentând numărul cutiilor. Următoarele *n* linii conțin fiecare câte un număr natural, reprezentând dimensiunile cutiilor.

Date de ieşire
Pe prima linie a fişierului **CUTII.OUT** se va scrie numărul minim *m* de pachete de cutii. Pe următoarele *m* linii se vor scrie dimensiunile cutiilor pe care Ştefan le va împacheta într-un singur pachet, în ordinea inversă împachetării.

Restricții
- $0 \le n \le 1000$;
- $1 \le dimensiune\_cutie \le 15000$.

Exemplu

CUTII.IN
```
10
4
1
5
10
7
9
2
8
3
2
```

CUTII.OUT
```
4
1 4
2 5
2 3 7 10
8 9
```

9.9. Soluțiile problemelor propuse

9.9.1. Polinom

Deoarece se poate întâmpla ca polinoamele să fie „rare”, (vor avea mulți coeficienți egale cu 0), cea mai potrivită reprezentare o vom realiza cu o listă alocată dinamic în care păstrăm câte un monom al polinomului. Un monom se descrie prin coeficientul său şi gradul necunoscutei *x*.

Se ştie că adunarea a două polinoame se realizează termen cu termen, dar nu putem lua simplu un monom din primul polinom şi unul din al doilea, deoarece se poate întâmpla ca gradele acestor monoame să difere, caz în care acestea nu se pot aduna. În concluzie, ne vom situa în dreptul primului monom în primul polinom şi vom aduna coeficientul acestui monom cu coeficientul din celălalt doar în cazul în care gradele celor două monoame sunt identice. În caz contrar vom avea un termen nou în polinomul sumă format fie din monomul respectiv al primului polinom (dacă gradul acestuia este mai mic decât gradul monomului celui de al doilea polinom) fie din al doilea. În polinomul sumă trebuie să avansăm în ordinea crescătoare a gradelor, deoarece va trebui să afişăm coeficienții polinomului sumă în ordine descrescătoare după gradul necunoscutei şi aceşti coeficienți se păstrează într-o listă simplu înlănțuită de tip LIFO, în cazul cărora prelucrarea inversează ordinea elementelor.

Declarațiile aferente celor două polinoame date şi polinomului sumă sunt următoarele:

```
type lista=^monom;      { lista este tipul pointerilor care vor referi monoame }
     monom=record
              coef:Integer;                      { coeficientul monomului }
              gradx:Byte;                          { gradul monomului }
              urm:lista       { câmpul care asigură înlănțuirea monoamelor }
           end;
var cap,cap1,cap2:lista;          { pointerii care indică primul monom din }
...                  { polinomul sumă, respectiv din primul şi al doilea polinom }
```

După ce s-au creat listele simplu înlănțuite corespunzătoare celor două polinoame date, calculăm polinomul sumă. Algoritmul este de fapt un algoritm de interclasare:

```
procedure Suma(var cap:lista; cap1,cap2:lista);
var p,q,r:lista;
begin
  p:=cap1;            { cap1 indică monomul de grad minim în primul polinom }
  q:=cap2;           { cap2 indică monomul de grad minim în al doilea polinom }
  cap:=nil;
```

```
  while (p <> nil) and (q <> nil) do begin
                                   { dacă avem termeni în ambele polinoame }
    New(r);                       { creăm element nou pentru polinomul sumă }
    if p^.gradx = q^.gradx then
      with r^ do begin
        coef:=p^.coef+q^.coef;    { gradele sunt egale, adunăm coeficienţii }
        if coef = 0 then
          Dispose(r)              { dacă avem coeficient nul, nu creăm element }
        else begin
          gradx:=p^.gradx;        { creăm elementul cu toate câmpurile sale }
          urm:=cap;
          cap:=r
        end;
        cap1:=cap1^.urm;
        Dispose(p);               { eliberăm spaţiul alocat elementelor prelucrate }
        cap2:=cap2^.urm;
        Dispose(q);
        p:=cap1;                  { repoziţionăm pointerii auxiliari pe capul listelor }
        q:=cap2
      end else begin { dacă în primul polinom, termenul este de grad mai mic }
        if p^.gradx < q^.gradx then begin
          Element_nou(cap,cap1,r,p)      { îl inserăm în polinomul sumă }
        else                      { altfel inserăm termenul din al doilea polinom }
          Element_nou(cap,cap2,r,q)
  end;
  while p <> nil do begin         { dacă mai există termeni în primul polinom }
    New(r);
    Element_nou(cap,cap1,r,p)
  end;
  while q <> nil do begin         { dacă mai există termeni în al doilea polinom }
    New(r);
    Element_nou(cap,cap2,r,q)
  end
end;
```

În această procedură se apelează subprogramul `Element_Nou(cap,cap1,r,p)` în care se creează câmpurile elementului nou `r^` a listei monoamelor polinomului sumă (pointerul `cap` indică primul element al acesteia) şi se eliberează spaţiul alocat monomului copiat în lista originală (referit de pointerul `cap1`). Pointerul `p` se repoziţionează pe primul element al listei originale.

```
procedure Element_Nou(var cap,cap1,r,p:lista);
begin
  with r^ do begin
    coef:=p^.coef;
    gradx:=p^.gradx;
    urm:=cap;
    cap:=r;
  end;
  cap1:=cap1^.urm;
  Dispose(p);
  p:=cap1
end;
```

9.9.2. Farfurii

Din farfuriile despachetate creăm o stivă alocată dinamic, apoi o parcurgem în scopul căutării farfuriei având modelul dat. În timpul căutării, farfuriile verificate le punem deoparte într-o stivă nouă, totodată numărându-le. În final, afişăm răspunsul la întrebare în funcție de rezultatul căutării şi afişăm conținutul celei de-a doua stive.

Dacă prima farfurie verificată (ultima despachetată) este cea căutată, stiva auxiliară este vidă. Dacă farfuria s-a spart şi nu o găsim în stivă, toate farfuriile despachetate se mută în stiva auxiliară.

9.9.3. Agenție

Cele n persoane înscrise formează o coadă de aşteptare. La excursie vor participa primii *k* dintre ei. Dacă s-au înscris mai puține persoane decât *k*, îi vom afişa pe toți care s-au înscris.

9.9.4. Mesaje

Rezultă din enunț că cea mai potrivită structură de date cu care am putea reprezenta lista prietenilor lui Vasile este o listă circulară. Inițial creăm o coadă (cu santinelă) apoi legăm ultimul element de primul:

```
...
  ultim^.urm:=santinela^.urm;
...
```

Urmează traversarea listei circulare urmând referințele *urm*, efectuând o numărătoare până la *k*. Fiecare al *k*-lea element se marchează. Traversarea listei se opreşte în momentul în care o numărătoare se termină pe un element marcat deja. În paralel, fiecare al *k*-lea element se scrie în fişier, acestea reprezentând numele prietenilor lui Vasile care au primit mesajul de felicitare.

9.9.5. Admitere

Va trebui să citim datele şi să construim o listă ordonată după medii, iar în caz de egalitate, după numele candidaţilor. În programul principal, după citirea numărului candidaţilor şi a numărului de locuri în anul I, creăm lista vidă, în care câmpul nume din santinelă va fi *string*-ul vid.

```
Begin
  Assign(f,fin); Reset(f);                    { deschidem fişierul de intrare }
  ReadLn(f,n,k);                                                   { citire }
  New(cap);                                                { creăm lista vidă }
  cap^.urm:=nil;
  cap^.num:='';
  cap^.medie:=0;
  for i:=1 to n do begin
    New(nou);                        { creăm un element, fără să îl legăm în listă }
    ReadLn(f,nou^.num);
    ReadLn(f,nou^.medie);
    nou^.confirmat:=false;             { deocamdată nu a confirmat nimeni }
    Inserare(cap,nou)                    { inserarea elementului nou în listă }
  end;
  Verif(cap);                                  { completăm câmpul confirmat }
  Close(f);
  Afiseaza(cap)       { afişăm candidaţii admişi (au confirmat şi media este ≥ 5) }
End.
```

În subprogramul `Inserare(cap,nou)` căutăm poziţia unde trebuie să inserăm elementul `nou^`:

```
procedure Inserare(var cap:lista; nou:lista);
          { inserăm elementul nou în lista ordonată după medii; cap este santinela; }
                    { candidaţii cu medii egale se inserează în ordine alfabetică }
var p:lista;
begin
  p:=cap;                                        { căutăm locul în funcţie de medie }
  while (p^.urm <> nil) and (p^.urm^.medie > nou^.medie) do
    p:=p^.urm;
  if p^.urm <> nil then                          { căutăm locul în funcţie de nume }
   while (p^.urm^.medie=nou^.medie)and(p^.urm^.num<nou^.num) do
     p:=p^.urm;
  nou^.urm:=p^.urm;                                          { inserăm elementul }
  p^.urm:=nou
end;
```

9.8.6. Pregătirea mesei

Problema a fost tratată în capitolul 5, unde s-a explicat metoda *sortării topologice*. Această metodă de sortare se poate implementa mai natural cu ajutorul unui şir în care fiecare element este un cap de listă. În rest algoritmul este cel prezentat în capitolul 5 (teoria grafurilor).

9.9.7. Sportivi

Această problemă o cunoaştem deja din capitolul 3 (*euristica greedy*). Vom păstra lista sportivilor care se antrenează la acelaşi aparat.

9.9.8. Cutii

Problema de fapt cere ca dintr-un şir de numere date într-o ordine oarecare să creăm număr minim de subşiruri strict descrescătoare. Strategia de abordare a rezolvării din punct de vedere algoritmic este *greedy*, dar avem câteva probleme privind spațiul de memorie necesar pentru structurile de date. De exemplu, dacă avem cazul particular în care şirul este dat în ordine crescătoare, fiecare cutie constituie un pachet, deci ar trebui să declarăm atâtea subşiruri câte numere avem în şirul dat. În plus, dacă avem cazul particular în care şirul este dat în ordine descrescătoare, avem un singur pachet, deoarece fiecare cutie nouă încape în cutia precedentă, deci subşirul ar trebui să fie declarat având aceeaşi lungime cu şirul dat. În enunț s-a specificat că pot exista 8000 de cutii care ar necesita un tablou bidimensional de 64.000.000 de elemente... Evident, nu putem declara un astfel de tablou. (Şi nu am luat în considerare faptul că în cazul subşirurilor ar trebui să păstrăm lungimea efectivă a fiecăruia etc.)

Să considerăm exemplul din enunț în care avem 10 cutii având dimensiunile: 4, 1, 5, 10, 7, 9, 2, 8, 3, 2. Prima cutie constituie totodată şi primul pachet.

Pachet 1
4

Încă nu ştim dacă vom pune sau nu altă cutie în ea, dar în momentul în care vine la rând cutia de dimensiune 1, datorită faptului că acesta încape în cutia de dimensiune 4, o punem în ea.

Pachet 1
4, 1

Urmează cutia de dimensiune 5. Aceasta nu încape într-o cutie de dimensiune 1, deci începem cu ea un pachet nou:

Pachet 1	Pachet 2
4, 1	5

Urmează cutia de dimensiune 10, care nu încape în nici una dintre cutiile aşezate, deci avem un treilea pachet:

Pachet 1	Pachet 2	Pachet 3
4, 1	5	10

Cutia următoare are dimensiunea 7 şi încape în ultima cutie aşezată:

Pachet 1	Pachet 2	Pachet 3
4, 1	5	10, 7

Acum trebuie să găsim loc pentru cutia având dimensiunea 9. Aceasta nu mai poate fi pusă în cea de dimensiune 10, pentru că aceasta este ocupată şi nu avem voie s-o eliberăm, deci începem al 4-lea pachet:

Pachet 1	Pachet 2	Pachet 3	Pachet 4
4, 1	5	10, 7	9

Urmează o cutie de dimensiune 2. Ea ar încape în pachetele 2, 3 şi 4, dar, pe baza strategiei greedy o vom pune acolo unde stă „cât mai strâmt", astfel evitând risipa de spațiu, adică în pachetul 2:

Pachet 1	Pachet 2	Pachet 3	Pachet 4
4, 1	5, 2	10, 7	9

Următoarea cutie încape în pachetul 4:

Pachet 1	Pachet 2	Pachet 3	Pachet 4
4, 1	5, 2	10, 7	9, 8

Penultima şi ultima cutie încap, una după alta în pachetul 3:

Pachet 1	Pachet 2	Pachet 3	Pachet 4
4, 1	5, 2	10, 7, 3, 2	9, 8

Din această execuție pas cu pas a algoritmului care trebuie implementat rezultă două observații foarte importante.

1. La fiecare pas şirul format din ultimele elemente ale subşirurilor este ordonat crescător;
2. Pentru a găsi locul cutiei curente este suficient să îi căutăm locul în acest şir.

Pe baza observațiilor decidem să ținem evidența acestor ultime cutii în vârful a câte unei stive, astfel mutând majoritatea datelor în heap, păstrând în memoria statică doar şirul pointerilor care indică ultima cutie aşezată în fiecare pachet. În plus, datorită faptului că aceste elemente din vârfurile stivelor sunt ordonate crescător, vom aplica algoritmul căutării binare pentru a găsi locul noii cutii.

În programul principal vom iniția stivele vide cu **nil**. Apoi, vom citi câte o dimensiune de cutie şi îi vom căuta locul cu subprogramul `Inserare(stanga, dreapta,dimnou)` care se apelează cu parametri `1`, `k` şi `dimnou`, unde `1` reprezintă indicele din stânga al şirului pachetelor, `k` este indicele din dreapta, iar `dimnou` este valoarea de inserat. La primul apel valoarea lui `k` (indicele din dreapta) este `0`, deoarece încă nu avem nici un pachet.

Subprogramul de inserare este următorul:

```
procedure Inserare(stanga,dreapta:Byte; dimnou:Word);
var mijloc:Word;
    p:pachet;
begin                                { nu am găsit nici un pachet corespunzător }
  if stanga > dreapta then begin
    Inc(dreapta);
    k:=dreapta;                                                  { pachet nou }
    New(p);                          { aşezăm cutia curentă în acest pachet nou }
    p^.dim:=dimnou;
    p^.urm:=pachete[k];
    pachete[k]:=p
  end else begin
    if pachete[stanga]^.dim > dimnou then begin
      New(p);              { am găsit un pachet în care încape cutia curentă }
      p^.dim:=dimnou;
      p^.urm:=pachete[stanga];
      pachete[stanga]:=p
    end else begin
      mijloc:=(stanga+dreapta) div 2;
      if dimnou < pachete[mijloc]^.dim then
        Inserare(stanga,mijloc,dimnou)
      else
        Inserare(mijloc+1,dreapta,dimnou)
    end
  end
end;
```

Dacă ar fi trebuit afişate şirurile exact în ordinea în care s-au împachetat cutiile, în loc de stive, am fi lucrat cu cozi, dar astfel ar fi trebuit să declarăm atât şirul vârfurilor cât şi şirul ultimelor elemente şi nu am fi putut prelucra 15000 de cutii.