

Tehnica backtracking

Tehnica backtracking admite o rezolvare lenta, dar de cele mai multe ori este singura rezolvare a anumitor probleme. În programare folosim această tehnică numai în situația în care nu disponem de o altă metodă de rezolvare. În problemele trebuie să îndeplinească simultan următoarele condiții:

- soluția poate fi pusă într-un vector
- elementele vectorului aparțin unor multimi; fiecare element aparține unei multimi (multimile la unele probleme pot coincide, iar aceste multimi sunt finite)

Pașii pe care îi urmăm pentru rezolvarea unei astfel de probleme sunt următorii:

Presupunem că am găsit k elemente din vectorul soluție și atunci vom căuta următorul element ($k+1$). Aceste elem. se va căuta în multimea A_{k+1} . Din acest punct avem 2 variante:

- a) în cazul în care nu găsim acest elem. în multimea corespunzătoare ne vom întoarce cu ~~pe~~ un pas în urmă și vom căuta un alt elem. x_k în multimea A_k .

b) am găsit elem. x_{k+1} și vom verifica dacă îndreptează condițiile. Dacă da, vom avea următoarele posibilități:

b₁. am găsit soluția, o tipărim și apoi căutăm un alt elem. din A_{k+1} până la testarea tuturor soluțiilor din multimea respectivă

b₂. nu am ajuns la soluție și vom căuta un alt elem. din multimea A_{k+2} .

Elementele vectorului le putem privi ca pe o stivă și astfel vom nota stiva cu st . Fiecare elem. al stivei se va initializa cu o valoare care nu existe în elem. vectorului. În general se va initializa cu un elem. care se află în fața tuturor elem. mulțimilor. Vom utiliza o funcție de initializare numită INIT. Pentru a căuta vom folosi funcția AM_SUCESOR. Dacă am găsit succesorul și vom testa dacă este valid.

La final, funcția SOLUȚIE verifică dacă am găsit soluția și e completă. În acesta, caz, aceasta se va tipări folosind funcția TIPAR.

Toate aceste funcții prezentate, formează tehnica backtracking. Îi toate vor fi înglobate într-o funcție generică numită backtracking.

```
void backtracking(int m){  
    int st[10], nivel_st=0;  
    bool as;  
    init(st, nivel_st);  
    while (nivel_st >= 0) {  
        while ((as == am_succesor(st, nivel_st, m)) &&  
               !e_valid(st, nivel_st));
```

```

if(as) {
    if(solutie(nivel_st,m)) {
        tipar(st,m);
    }
    else {
        nivel_st++;
        init(st,nivel_st);
    }
}
else
    nivel_st--;

```

Problema celor m regine:

Se dă o tablă de sah de dimensiune $m \times n$. Se cer toate posibilitățile de aranjare a m regine, astfel încât reginile să nu se atace (să nu se afle 2 regine pe aceeași linie, coloană sau diagonală).

Pentru rezolvarea problemei procedăm în felul următor:

- INIT - nivelul k este initializat cu valoarea 0
- AM-SUCESOR: măreste cu o unitate continutul stivii pe nivelul k , având grija să nu se depășească valoarea m (în funcție de această condiție parametrul AS ia valoarea true sau false)
- VALID: verifică valoarea dată de funcția anterioară pe nivelul k al stivii având grija să fie îndeplineite cele două condiții în care reginile nu se atacă: pe aceeași coloană $st(i) = st(k)$ și pe aceeași diagonală: $Abs(st(i)-st(k)) = Abs(i-k)$ (în funcție de aceste condiții parametrul EV ia valoarea true sau false)

→ SOLUȚIE: verifică dacă sirul a fost completat
prin să le nivelelui m

→ TIPAR: tipărește o soluție

#include <iostream>

#include <cmath>

using namespace std;

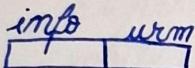
Liste liniare simple înlățuită (LLS)

• lista liniară este o colecție de elemente care se află

În lista simplu înlățuită fiecare elem. memorază două informații:

- informația utilă (info)

- informația de legătură către următorul element (urm)



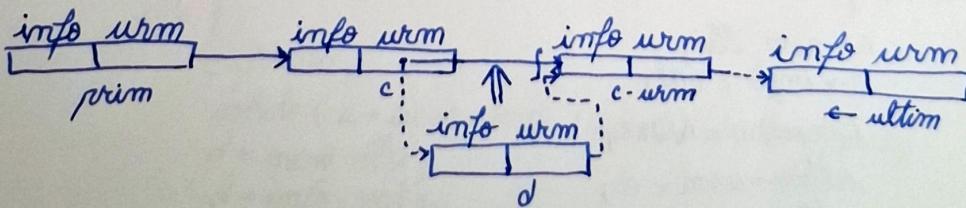
```
struct llsi {
    int info;
    llsi *urm;
}
```

} Definiția structurii

Operări care se pot efectua:

1. crearea listei
2. parcurgerea listei
3. adăugarea unei elem. la început, la sfârșit sau în interiorul listei
4. stergerea unui elem. din listă

Încrerarea unui elem. în listă:



Variabile:

- prim, ultim
- urm
- c : pointer către elem. curent
- info : informația elementului din lista
- d : pointer către nouă elem. introdus sau sters
- val :
- val : valoarea din lista înaintea căreia sau după ce inseră un elem.

```
#include <iostream>
using namespace std;
struct llii{
    int info;
    llii *urm;
};

lli *prim, *ultim;
void creare (lli *prim, int val){
    lli *c;
    if (!prim) {
        prim = new lli;
        prim->info = val;
        prim->urm = NULL;
        ultim = prim;
    }
    else {
        c = new lli;
        c->info = val;
        c->urm = NULL;
        ultim->urm = c;
        ultim = c;
    }
}
```

3

```
void listare (llsi *prim) {
    llsi *c;
    c = prim;
    while (c) {
        cout << c->info << " ";
        c = c->urm;
    }
}
```

```
void inserire (llsi *prim, int val, int val1) {
    llsi *c, *d;
    c = prim;
    while (c->info != val)
        c = c->urm;
    d = new llsi;
    d->info = val1;
    d->urm = c->urm;
    c->urm = d;
}
```

```
void inserireante (llsi *&prim, int val, int val1) {
    llsi *c, *d;
    if (prim->info == val) {
        d = new llsi;
        d->info = val1;
        d->urm = prim;
        prim = d;
    } else {
        c = prim;
        while (c->urm->info != val) c = c->urm;
        d = new llsi;
        d->info = val1;
        d->urm = c->urm;
        c->urm = d;
    }
}
```

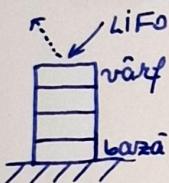
```
void stergere (lls *prim, int val) {
    lli *c, *aux;
    c = prim;
    if (c->info == val) {
        prim = c->urm;
        delete c;
    }
    else {
        c = prim;
        while (c->urm->info != val) c = c->urm;
        aux = c->urm;
        c->urm = aux->urm;
        delete aux;
    }
}
```

Stiva și coada

Atât stiva cât și coada sunt liste liniare speciale.

Stiva este o listă liniară prin care elementele se adaugă și se extrag printr-un singur capăt numit vârf stivei.

Prințul element introdus în stivă s.m. baza stivei.



Operări:

- adăugarea de elemente
- parcurgerea stivei
- eliminarea elementelor din stivă

Declarație stivei:

```
struct stivă {
    int info;
    stivă *pre; } stivă *v;
```

Coada este o listă liniară simplu înăntuită specială la care elementele se adaugă la final și se elimină la început.
→ același 3 operații

Declarație:

```
struct coadă {
    int info;
    coadă *urm; } coadă *cap, *sf;
```

Lista liniară dublu înăntuită (LLDl)

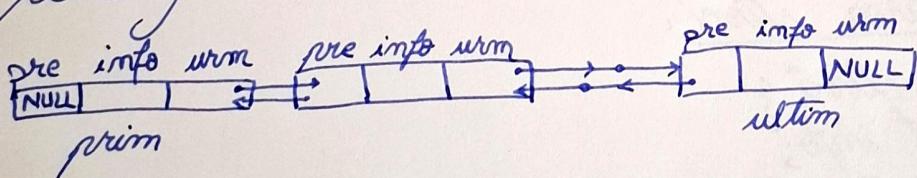
În LLDl fiecare element memorează 3 informații: informația utilă (info), informația de legătură către urm. elem. din listă (urm) și informația de leg. cu elem. precedent din listă (pre)

! Definire în C++:

```
struct LLDl {  
    int info;  
    LLDl *pre, *urm;};
```

Operări:

- crearea listei
- adăugarea de elem. în listă (în față, în mijloc, sau în interior)
- stergerea unui elem. din listă
- parcurgerea listei în cele două sensuri



Listele circulare pot fi simplu sau dublu înăntuite în care ultimul element e legat de primul elem.