

Statikus adatszerkezetek

Szerző: Horváth Gyula
Előadó: Németh Zsolt

2025. szeptember 11.

Feladat: Baráti lehallgatás

Kiki nagyon féltékeny a barátjára, ezért úgy cselekszik, ahogy minden érett fiatal felnőtt tenné a helyében: keylogger vírust telepít a barátja számítógépére. A vírus naplózza a leütött billentyűket és Kiki szeretné ezekből a gombnyomásokból visszaállítani a barátja üzeneteit. Az egyszerűség kedvéért tegyük fel, hogy az üzenetek írása során Kiki barátja csak az angol ábécé kisbetűit és a számjegyeket írta be, és ezeken kívül a balra (L) és jobbra (R) nyilakat, valamint a backspace (B) visszatörlő gombot használta. Írj programot, ami a gombnyomások sorozatából visszaállítja Kiki barátjának titkos üzenetét!

Bemenet

A standard bemenet első és egyetlen sorában egy S karakterlánc található. Jelölje N a karakterlánc hosszát. Mind az N karakter az angol ábécé egy kisbetűje, egy számjegy, vagy az L, R és B karakterek egyike. Ha a kurzor épp a szöveg elején van, akkor biztosan nem fog L vagy B következni, továbbá ha a kurzor a szöveg végén van, akkor nem fog R következni

Kimenet

A standard kimenetre egy sort kell írni, a Kiki barátja által írt üzenetet szövegét.

Példa

Bemenet

aLiLlLuLj

nevetlekLLLLBBB

password1234

Kimenet

julia

szeretlek

password1234



A megoldás algoritmusának kifejlesztéséhez mondjuk meg, hogy milyen adatokon milyen műveleteket kell végezni.

A bemenet egy karaktersorozat, amelynek minden eleme egy műveletet határoz meg.

Minden művelet egy (karakter) sorozaton végez műveletet.

Kezdetben a sorozat üres. Ha minden bemenetbeli karakterre ismerjük az elvégzendő műveletet, akkor a megoldás algoritmusa a következőképpen adható meg.

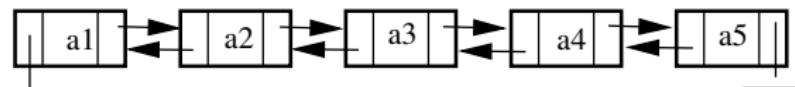
```
1 Sorozat S=Sorozat(); //üres sorozat létesítés
2 string bemenet;
3 getline(cin, bemenet);
4 for(int i=0;i<bemenet.size();i++){
5     char x=bemenet[i];
6     switch(x){
7         case 'L': S.balra(); break;
8         case 'R': S.jobbra(); break;
9         case 'B': S.torol(); break;
10        default : S.beszur(x);
11    }
12 }
13 S.kiir();
```

A felhasznált Sorozat-műveletek többféleképpen valósíthatók meg. Az aktuális sorozatot tárolhatjuk egy tömbben folytonosan, de ekkor a beszúrás és a törlés nem lesz hatékony.

A sorozatok ábrázolására olyan megoldást, adatszerkezetet keresünk, amely lehetővé teszi, hogy minden művelet konstans futási idejű lehet.

Tároljuk a sorozatot kétirányú láncban. ez olyan adatszerkezet, amely cellák (memória-helyek) sorozatából áll, minden cella három érteket tartalmaz

1. adatot, a sorozat egy elemét,
2. az előző sorozatelemet tartalmazó cellára való hivatkozást,
3. a következő sorozatelemet tartalmazó cellára való hivatkozást.



A $(a_1, a_2, a_3, a_4, a_5)$ sorozatot ábrázoló kétirányú lánc.

Ez még nem konkrét adatszerkezet, mert nem tudjuk, hogy a celláknak hogyan foglalunk helyet és a két kapcsolatot hogyan adjuk meg.

Alapvetően két módszer ismert:

statikus

1. minden cella számára egy tömb, vagy dinamikus tömb elemeként foglalunk memória helyet.
2. A referencia ekkor a cellát tartalmazó tömbelem indexe.

dinamikus

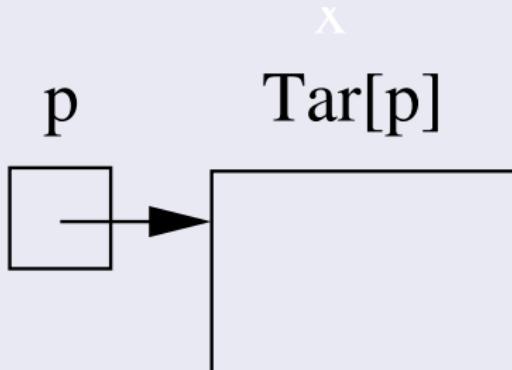
1. A cellák számára dinamikusan (végrehajtáskor) a **new** művelettel foglalunk memóriát.
2. A szerkezeti kapcsolatokat cellára mutató hivatkozással (**pointer**) valósítjuk meg.



Hivatkozás statikus ábrázolás esetén

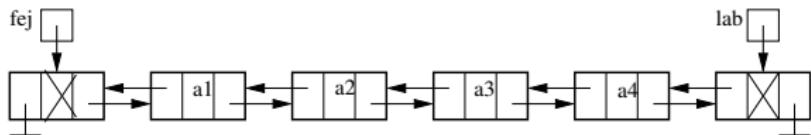
Tegyük fel, hogy a cellák számára **Cella Tar[maxN]** tömbben foglalunk memóriát.

Ekkor a **p** referencia típusú változó, akkor az általa hivatkozott cella a **Tar[p]**



A kétirányú lánc ábrázolása

A műveletek egységes kezelése végett célszerű a lánc elejére és végére egy-egy (üres) cellát tenni, amely nem tartalmaz sorozat elemet.



Az (a_1, a_2, a_3, a_4) sorozatot ábrázoló kétirányú lánc **fej** és **lab** cellával.

A kétirányú lánc statikus adatszerkezet

```
1 struct Cella;
2 typedef int Ref;
3 const int NIL=-1; //nemlétező cella referenciája
4 typedef char E; //a sorozat elemeinek a típusa
5 struct Cella{
6     E adat;
7     Ref elozi, kovet;
8 };
9 Cella Tar[maxM]; // memória a cellák számára
10 int szabad=1; //az első szabad cella indexe
11 Ref UjCella(){ //új cella kérése a táróból
12     int idx=szabad;
13     szabad=Tar[szabad].kovet;
14     return idx;
15 }
16 void VisszaAd(Ref c){ //a c cella megy a szabadok közt
17     Tar[c].kovet=szabad;
18     szabad=c;
19 }
```

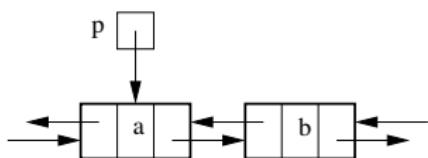
A kétirányú lánc adatszerkezet

A kétirányú lánc adatszerkezet

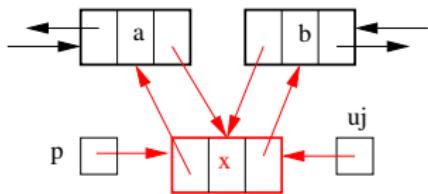
```
20 struct Lanc{  
21     Ref fej;  
22     Ref lab;  
23     Lanc(){  
24         fej=UjCella(); lab=UjCella();  
25         Tar[fej].eloz=NIL; Tar[fej].kovet=lab;  
26         Tar[lab].eloz=fej; Tar[lab].kovet=NIL;  
27     }  
28 };  
29 void Kezd(){//a szabad cellák összeláncolása  
30     for(int i=1;i<maxM-1;i++)  
31         Tar[i].kovet=i+1;  
32     Tar[maxM-1].kovet=NIL;  
33 }
```

A Lancba művelet megvalósítása

```
1 Ref Lancba( E x, Ref p){  
2     if(p==lab) return NIL;  
3     Ref uj=UjCella();  
4     Ref kov=Tar[p].kovet;  
5     Tar[p].kovet=uj;  
6     Tar[uj].adat=a; Tar[uj].eloz=p; Tar[uj].kovet=kov;  
7     Tar[kov].eloz=uj;  
8     return uj;  
9 }
```



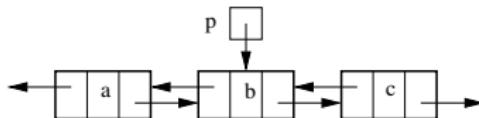
A művelet előtt.



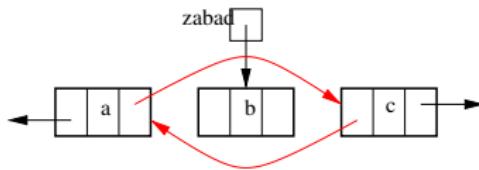
A művelet után.

A Cellatorol művelet megvalósítása

```
1 Ref Torol( Ref p){  
2     if (p==fej || p==lab) return NIL;  
3     Ref e=Tar[p].eloz;  
4     Ref u=Tar[p].kovet;  
5     Tar[e].kovet=u;  
6     Tar[u].eloz=e;  
7     VisszaAd(p);  
8     return u;  
9 }
```



A művelet előtt.



A művelet után.

```
1 struct Sorozat{  
2     Ref fej, lab, kurzor;  
3     Sorozat(){ //konstruktor  
4         fej=UjCella(); lab=UjCella();  
5         Tar[fej].kovet=lab; Tar[fej].eloz=NIL;  
6         Tar[lab].kovet=NIL; Tar[lab].eloz=fej;  
7         kurzor=fej;  
8     }  
9     // A műveletek deklarációja  
10    void balra();  
11  
12    void jobbra();  
13  
14    void beszur(E x);  
15  
16    void torol();  
17  
18    void Felsorol(void muvel(E));  
19};
```

```
1 void Sorozat:: balra(){
2     if(kurzor!=fej) kurzor=Tar[kurzor].eloz;
3 }
4 void Sorozat:: jobbra(){
5     if(kurzor!=lab) kurzor=Tar[kurzor].kovet;
6 }
7 void Sorozat:: beszur(E x){
8     Lancba(x, kurzor);
9 }
10 void Sorozat:: torol(){
11     cellatorol(kurzor);
12 }
13 void Sorozat:: Felsorol(void muvel(E)){
14     Ref p=Tar[fej].kovet;
15     while(p!=lab){
16         muvel(Tar[p].adat); p=Tar[p].kovet;
17     }
18 }
```

```
1 int main(){
2     Sorozat S=Sorozat(); //üres sorozet létesítés
3     string bemenet;
4     getline(cin, bemenet);
5     for(int i=0;i<bemenet.size();i++){
6         char x=bemenet[i];
7         switch(x){
8             case 'L': S.balra(); break;
9             case 'R': S.jobbra(); break;
10            case 'B': S.torol(); break;
11            default : S.beszur(x);
12        }
13    }
14    Ref p=Tar[S.fej].kovet;
15    while(p!=S.lab){
16        cout<<Tar[p].adat;
17        p=Tar[p].kovet;
18    }cout<<endl;
19 }
```

Hatékonyúság

A nyilvánvaló, hogy (a Felsorol kivételével) minden művelet futási ideje konstans.

A memória felhasználás mértéke arányos a végrehajtott **Lancba** műveletek számával, ha nem gondoskodunk a törölt cellák újrahasznosításával .

Ezt lehet csökkenteni, ha a törölt cellák újrahasznosításával. Az újrahasznosítás megoldható úgy, hogy a még szabad cellákat láncba szervezzük, és új cella igény esetén ebből a láncból veszünk ki, a töröltet pedig ezen lánc elejére rakjuk.

```
1 Ref UjCella(){//új cella kérése a táróból
2     int idx=szabad;
3     szabad=Tar[szabad].kovet;
4     return idx;
5 }
6 void VisszaAd(Ref c){//a c cella visszakerül a szab
7     Tar[c].kovet=szabad;
8     szabad=c;
9 }
```

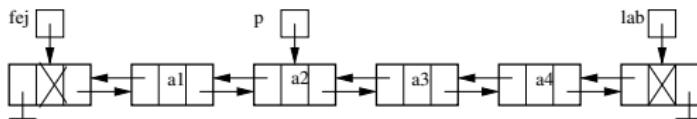
További lánc-műveletek

A továbbiakban a kétirányú láncot önálló objektumnak tekintjük, és megadjuk a műveleteit.

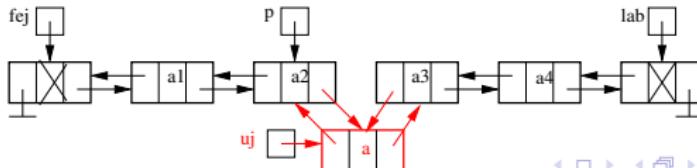
```
1 struct Lanc{  
2     Ref fej;  
3     Ref lab;  
4     Lanc(){  
5         fej=UjCella(); lab=UjCella();  
6         Tar[fej].eloz=NIL; Tar[fej].kovet=lab;  
7         Tar[lab].eloz=fej; Tar[lab].kovet=NIL;  
8     }  
9 };  
10 void Kezd(){ //a szabad cellák összelánkolása  
11     for(int i=1;i<maxM-1;i++)  
12         Tar[i].kovet=i+1;  
13     Tar[maxM-1].kovet=NIL;  
14 }
```

Lancba művelet

```
1 Ref Lancba(Lanc &L, Ref p, E a){  
2     if (p==L.lab) return NIL;  
3     Ref uj=UjCella();  
4     Ref kov=Tar[p].kovet;  
5     Tar[p].kovet=uj;  
6     Tar[uj].adat=a;  
7     Tar[uj].eloz=p;  
8     Tar[uj].kovet=kov;  
9     Tar[kov].eloz=uj;  
10    return uj;  
11 }
```

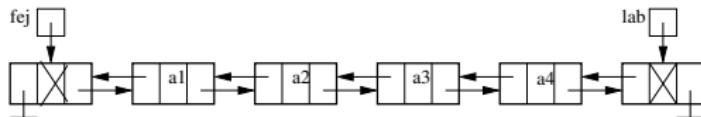


A művelet előtt

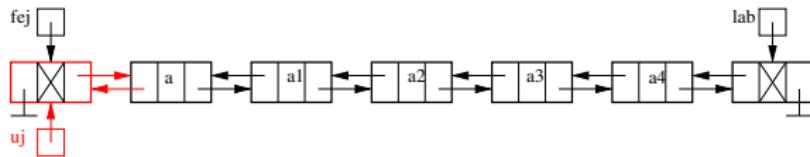


Elejére művelet

```
1 void Elejere(Lanc &L, E a){  
2     L.fej=Lancba(L, L.fej, a);  
3 }
```



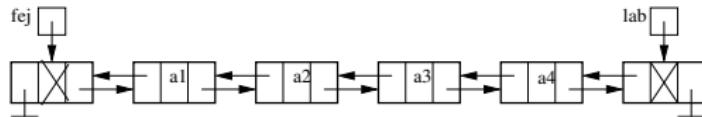
A művelet előtt



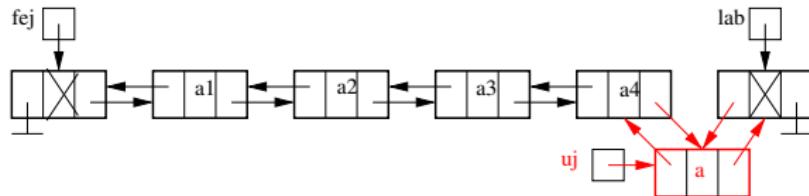
A művelet után

Végére művelet

```
1 void Vegere(Lanc &L, E a){  
2     Lancba(L, Tar[L.lab].eloz, a);  
3 }
```



A művelet előtt

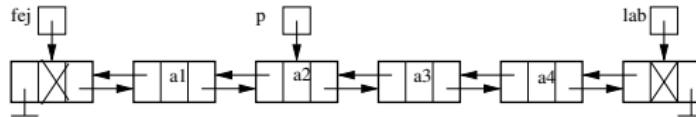


A művelet után

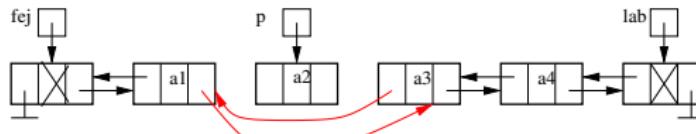
Töröl művelet

A visszaadott érték a törölt cellát követő cellára való hivatkozás.

```
1 Ref Torol(Lanc &L, Ref p){  
2     if (p==L.fej || p==L.lab) NIL;  
3     Ref e=Tar[p].eloz;  
4     Ref u=Tar[p].kovet;  
5     Tar[e].kovet=u;  
6     Tar[u].eloz=e;  
7     VisszaAd(p);  
8     return u;  
9 }
```



A lánc a művelet előtt

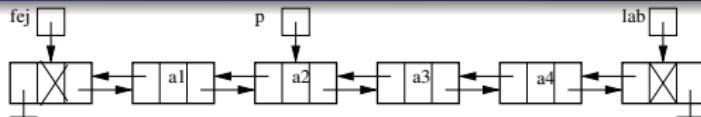


A lánc a művelet után

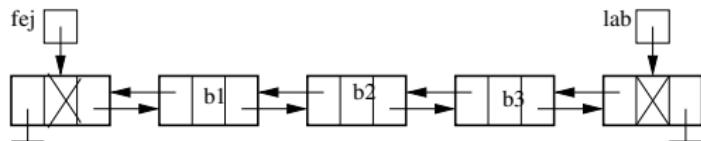
Keres és Üresít művelet

```
1 Ref Keres(Lanc &L, E a){
2     Ref p=L.fej;
3     while(p!=L.lab && a!=Tar[p].adat)
4         p=Tar[p].kovet;
5     if(p==L.lab) return NIL; else return p;
6 }
7
8 Uresit(Lanc& L){
9 //ha visszaadjuk a törölt cellákat
10    Ref p=L.fej;
11    while(p!=L.lab){
12        Ref pk=Tar[p].kovet;
13        VisszaAd(p);
14        p=pk;
15    }
16
17    Tar[L.fej].kovet=L.lab;
18    Tar[L.lab].eloz=L.fej;
19 }
```

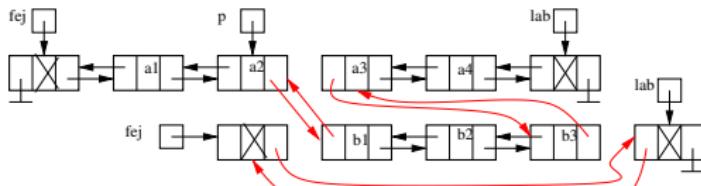
Láncba láncot beszúr művelet



Az *La* lánc a művelet előtt



Az *Lb* lánc a művelet előtt



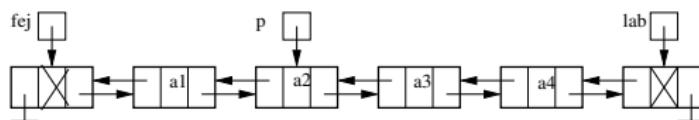
Az *La* lánc a művelet után. Az *Lb* lánc ües lesz.

Láncba láncot beszúr művelet

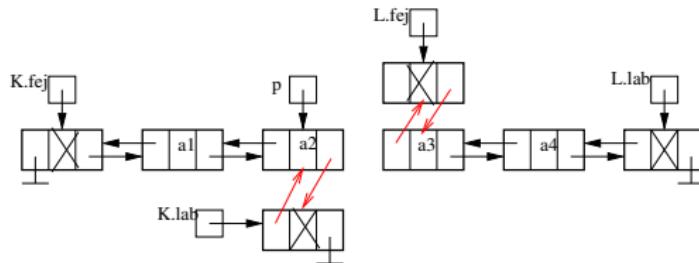
```
1 void LancbaLanc(Lanc& La, Ref p, Lanc& Lb){  
2     if (La.lab==p) return;  
3     if (Tar[Lb.fej].kovet==Lb.lab) return;  
4     Ref be=Tar[Lb.fej].kovet;  
5     Ref bu=Tar[Lb.lab].eloz;  
6     Tar[bu].kovet=Tar[p].kovet;  
7     Tar[Tar[p].kovet].eloz=bu;  
8  
9     Tar[p].kovet=be;  
10    Tar[be].eloz=p;  
11    Tar[Lb.fej].kovet=Lb.lab;  
12    Tar[Lb.lab].eloz=Lb.fej;  
13 }
```

Vág művelet

```
1 void Vag(Lanc& K, Ref p, Lanc& L){  
2     if(p==K.lab) return;  
3     L=Lanc();  
4     Tar[L.fej].kovet=Tar[p].kovet;  
5     Tar[Tar[p].kovet].eloz=L.fej;  
6     swap(K.lab, L.lab);  
7     Tar[p].kovet=K.lab;  
8     Tar[K.lab].eloz=p;  
9 }
```



A K lánc a művelet előtt



A K és L lánc a művelet után

Egyesít és Klon műveletek

```
1 void Egyesit(Lanc& L1, Lanc& L2){  
2     LancbaLanc(L1, Tar[L1.lab].eloz, L2);  
3 }  
4  
5 Lanc Klon(Lanc& L){  
6     Lanc K=Lanc();  
7     Ref p=Tar[L.fej].kovet;  
8     while(p!=L.lab){  
9         Vegere(K, Tar[p].adat);  
10        p=Tar[p].kovet;  
11    }  
12    return K;  
13 }
```

A Felsorol művelet

A **Felsorol** művelet a láncossal ábrázolt sorozat minden elemére sorrendben végrehajtja a paraméterként megadott **Muvel** műveletet.

```
1 void Felsorol(Lanc& L, void Muvel(E)){  
2     Ref c=Tar[L.fej].kovet;  
3     while(c!=L.lab){  
4         Muvel(Tar[c].adat);  
5         c=Tar[c].kovet;  
6     }  
7 }
```

Az első elem indexe 0.

```
1 Ref IndexElem(Lanc L, int i){  
2     Ref p=Tar[L.fej].kovet;  
3     while(p!=L.lab &&i>0){  
4         p=Tar[p].kovet;  
5         i--;  
6     }  
7     if(p==L.lab) return NIL; else return p;  
8 }
```

A műveletek futási ideje

Az **IndexElem** művelet futási ideje arányos az i paraméterrel.

A **Keres** művelet futási ideje legrosszabb esetben a lánc hosszával lesz arányos.

A **Klon** és a **Felsorol** művelet futási ideje minden esetben a lánc hosszával lesz arányos arányos.

Az **Uresít** művelet futási ideje minden esetben a lánc hosszával lesz arányos arányos ha visszaadjuk a lánc celláit.

A többi művelet futási ideje konstans.

1. Probléma szintje
2. Modell szintje
3. Absztrakt adattípus szintje
4. Absztrakt adatszerkezet szintje
5. Adatszerkezet szintje
6. Gépi szint

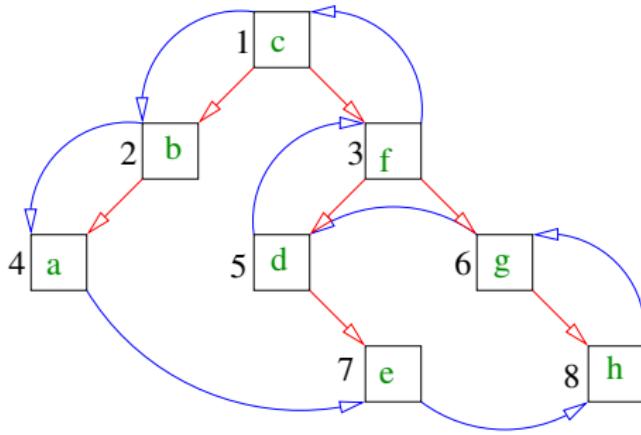
Adattípus: $A = (E, M)$

1. E : értékhalmaz,
2. M : műveletek halmaza.

Absztrakt, ha:

- i. nem ismert az adatokat tároló adatszerkezet,
- ii. nem ismertek a műveleteket megvalósító algoritmusok, a műveletek specifikációjukkal definiáltak.

Példa absztrakt adatszerkezetre



Adatszerkezet piros és kék szerkezeti kapcsolattal

$$M = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

kék : $M \rightarrow M$

piros : $M \rightarrow (M \cup \{\perp\}) \times (M \cup \{\perp\})$

adat : $M \rightarrow \text{char}$

A szerkezeti kapcsolatok megadása

cella	1	2	3	4	5	6	7	8
kék	2	4	1	7	3	7	8	6
piros	(2,3)	(4, ⊥)	(5,6)	(⊥, ⊥)	(⊥,7)	(⊥,8)	(⊥, ⊥)	(⊥, ⊥)
adat	c	b	f	a	d	g	e	h



Absztrakt adatszerkezet

Adatszerkezet: $S = (M, R, Adat)$

1. M az absztrakt memóriahelyek, *cellák* halmaza.
2. $R = \{r_1, \dots, r_k\}$ a cellák közötti szerkezeti kapcsolatok,
 $r_i : M \rightarrow (M \cup \{\perp\})^*$
3. $Adat : M \rightarrow E$ parciális függvény, a cellák adattartalma.

A \perp jelentése: nem létező szerkezeti kapcsolat.

$x \in M$, $r \in R$ és $r(x) = \langle y_1, \dots, y_i, \dots, y_k \rangle$

esetén az x cella r kapcsolat szerinti szomszédjai $\{y_1, \dots, y_k\}$,
 y_i pedig az x cella i -edik szomszédja.

Ha $y_i = \perp$, akkor azt mondjuk, hogy x -nek hiányzik az r szerinti
 i -edik szomszédja.

Absztrakt, ha:

- i. nem ismert, hogy a celláknak hogyan foglalunk memóriát,
- ii. nem ismert, hogy a szerkezeti kapcsolatokat hogyan valósítjuk meg.

Feladat: várótermi sorbanállás kezelése.

Probléma szintje

Kezelésre érkező betegek sorbanállását kell kezelní. A betegeket az TAJ számukkal azonosítják.

A következő funkciókat kell megvalósítani:

1. Beteg felvétele a sor végére.
2. Beteg felvétele sor elejére.
3. Beteg keresése a sorban.
4. Beteg törlése a sorból.
5. Beteg beszúrása sorba.
6. Sorban-állók listázása.

Modell szintje

Minden várakozási sor megadható a matematikai sorozat fogalmát használva.

$Adatsor = (b_1, b_2, \dots, b_n)$

Absztrakt adattípus szintje

Műveleteket definiálunk sorozatokon, amely műveletek a kívánt funkciókat valósítják meg.

Ezen a specifikációját adjuk meg, azt nem, hogy hogyan tároljuk az adatokat (sorozatokat), és milyen algoritmus valósítja meg a műveletet. Pl.

Vegere (S , b): a b beteg adatait (TAJ sorszámát) az S sorozat végére teszi.

Absztrakt adatszerkezet szintje

Megadjuk a sorozatok tárolásának módját. Például láncban tároljuk a sorozatot.

Olyan adatszerkezetet választunk, amely lehetővé teszi a megvalósítandó műveletek algoritmusának (hatékony) megvalósítását.

Konkrét adatszerkezet szintje

Megadjuk a sorozatok tárolásának módját, például láncolással.

Megadjuk, hogy a lánc elemeknek hogyan foglalunk memóriát, és ezzel azt is, hogy a szerkezeti kapcsolatokat (a következő lánc elemre hivatkozást) hogyan valósítjuk meg.

Feladat: várótermi sorbanállás kezelése.

Gépi szint

A fordító program megoldja.

Feladat: Leggyakoribb kezdőszelet (Prefixek).

Kutatók szövegelemzést végeznek. Összegyűjtöttök sok számukra érdekes szót. Van egy speciális szavakat tartalmazó gyűjteményük is, ezek a szavak nem szerepelnek az érdekes szavak között.

Szeretnék megtudni minden speciális szóra, hogy hány olyan érdekes szó van, amelynek ez kezdőszelete.

Bemenet

A bemenet első sora az összegyűjtött szavak N számát és a kérdések K számát tartalmazza. A további N sor mindegyike egy érdekes szót tartalmaz. A további K sor mindegyike egy speciális szót tartalmaz. Mindkét halmazbeli szavak csak az angol ábécé kisbetűit tartalmazzák.

A szavak összhossza legfeljebb 10^6 .

Kimenet

A kimenetre K sort kell írni, az i -edik sorba az i -edik kérdésre adandó választ kell írni!

Példa

Bemenet

9 2

fal

falu

far

tekla

tele

tere

teke

vele

vet

fa

te

Kimenet

3

4

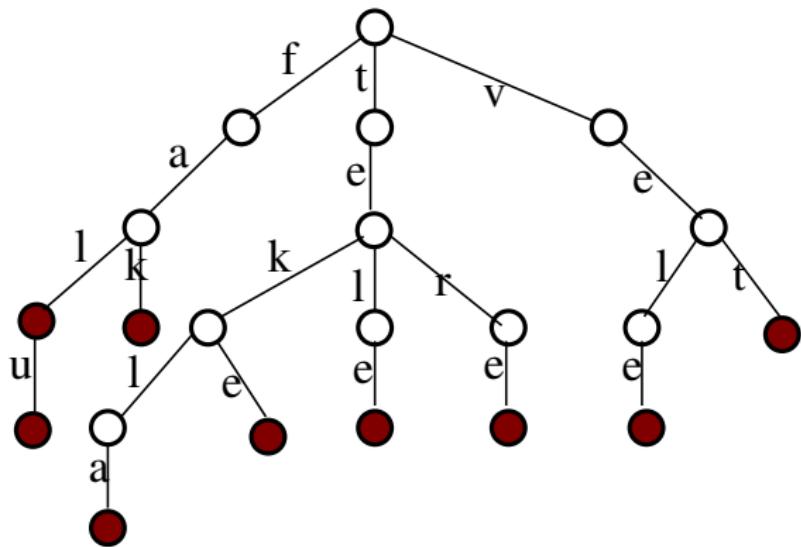
A naiv megoldás hatékonysága

Ha minden speciális szót minden érdekes szóval érdekes szóval összehasonlítjuk, akkor a futási idő legrosszabb esetben $O(\sum_{i=1}^K N * h_i)$ lesz, ahol h_i az i -edik speciális szó hossza.

Szófa (Trie, radixfa) absztrakt adatszerkezet

A szavak, mint stringek halmazát ábrázoljuk olyan fában, amelyre teljesül, hogy minden elágazás egy karakterrel van címkézve. A fára teljesül, hogy minden halmazbeli S szóhoz van olyan p pontja a fának, hogy a gyökértől a p -ig vezető úton lévő címkék sorozata megegyezik S -el, és ez a fa p pontjában jelezve van. A fa minimális, abban az értelemben, hogy nincs olyan részfája, amelyre az előbbi feltétel teljesülne.

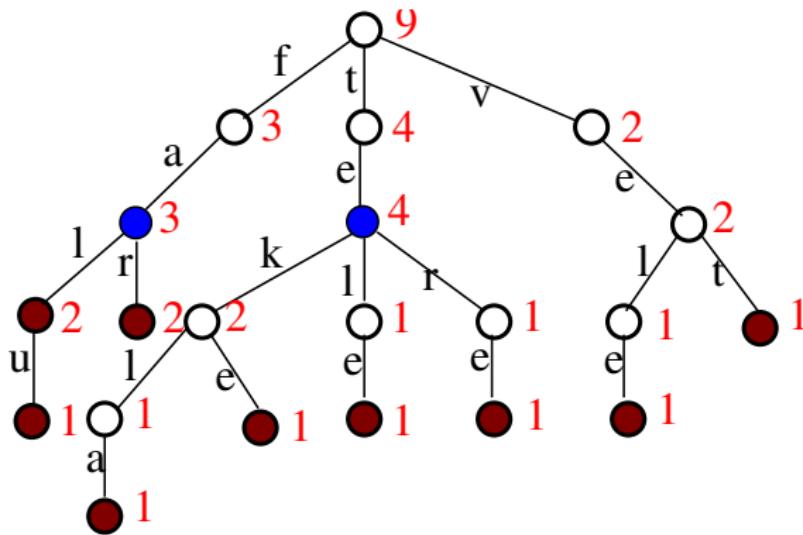
A példa bemenetet reprezentáló szófa



A piros pontokig vezető utak alkotják a reprezentálandó halmaz elemeit.

Megoldás

Adott **S** speciális szóra megoldás a fában azon piros pontok száma, amelyek az **S** szót reprezentáló pont-gyökerű részfában vannak.



A kék pontok reprezentálják a kérdéses speciális szavakat.

string elemeket tartalmazó halmaz alapműveletek megvalósítása Szófa adatszerkezettel

A három alapművelet:

1. **Faba(F , s)** : beilleszti az s stringet az F fába.
2. **Keres(F , s)** : az s -et reprezentáló fa pontjára mutató referenciát adja meg.
3. **Torol(F , s)** : törli s -et a fából, azaz az s -et reprezentáló fapont jelzettségét megszünteti.

Mindhárom alapművelet futási ideje s hosszával arányos idejű lesz.

Szófa adatszerkezet statikus ábrázolása

```
1 struct FaPont;
2 typedef int SzoFa; //tömbelem indexe
3 #define NIL -1 //nemlétező cella referenciája
4 typedef char E; //a sorozet elemeinek a típusa
5 const int maxM=100000; //tárméret a cellák
6 // (fapontok) számára
7 int szabad=1; //az első szabad cella indexe
8
9 typedef struct FaPont{
10     int van;
11     SzoFa Fiuk[26];
12     FaPont(){
13         van=false;
14         for (int i=0;i<26;i++) Fiuk[i]=NIL;
15     }
16 };
17 FaPont Tar[maxM]; // memória a cellák számára
```

Szófa alapművelet: Faba

```
1 Szofa UjFapont(){
2     int uj=szabad++;
3     Tar[uj].van=false;
4     for(int i=0;i<26;i++) Tar[uj].Fiuk[i]=NIL;
5     return uj;
6 }
7 void Faba(Szofa& F, string szo){
8     Szofa p=F;
9     for(int i=0; i<szo.length(); i++){
10         int idx=szo[i]-'a';
11         if (Tar[p].Fiuk[idx]==NIL){//uj fapont kérés
12             Tar[p].Fiuk[idx]=UjFapont();
13         }
14         p= Tar[p].Fiuk[idx]; //tovább az i-edik fiú felé
15     }
16     Tar[p].van=true; //sz bekerült a halmazba
17 }
```

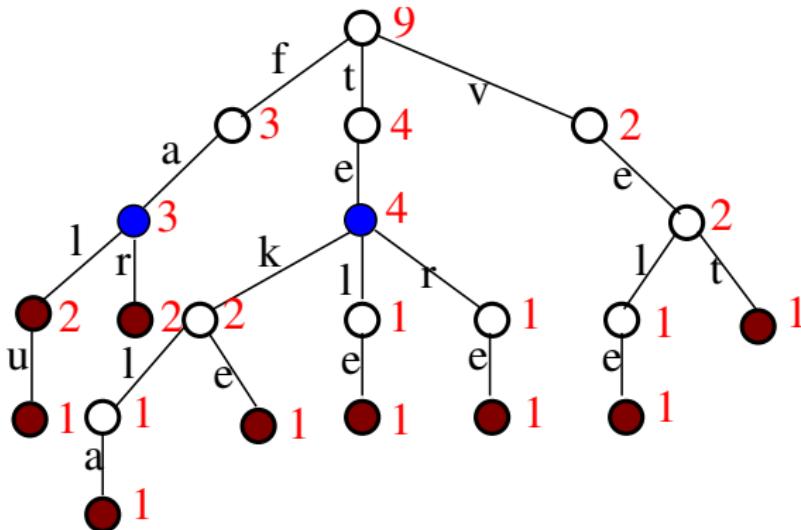
```
1 SzFa Keres(SzFa& F, string szo){
2     SzFa p=F;
3     for(int i=0; i<szو.length(); i++){
4         int idx=sزو[i]-'a';
5         if(Tar[p].Fiuk[idx]==NIL) return NIL;
6         p=Tar[p].Fiuk[idx];
7     }
8     return p;
9 }
10 bool Eleme(Szofa& F, string szo){
11     Szofa p=Keres(F, szo);
12     return p!=NIL && Tar[p].van;
13 }
```

Torol művelet

```
1 bool Torol(SzoFa& F, string szo){
2     SzoFa p=Keres(F, szo);
3     if(p!=NIL){
4         Tar[p].van=false;
5         return true;
6     }
7     return false;
8 }
```

A Prefixek feladat 1. megoldása

Vegyük észre, hogy a **Faba()** művelet algoritmusa egyszerűen módosítható úgy, hogy végül minden fapontban ott legyen az a szám, ami az adott gyökerű részfában lévő piros pontok száma. Felveszünk minden fapontba egy számlálót, aminek az értékét növeljük, ha a beszúráskor érintjük a pontot.



A számláló beszúrásokkal előállítható.

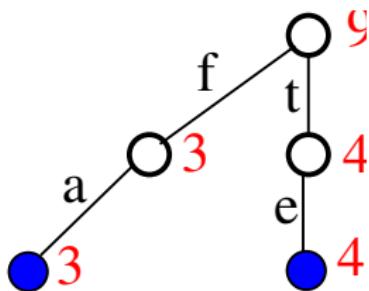
```
1 struct FaPont;
2 typedef int SzoFa; //tömbelem indexe
3 #define NIL -1 //nemlétező cella referenciája
4 typedef char E; //a sorozet elemeinek a típusa
5 const int maxM=1000001;
6 int szabad=1; //az első szabad cella indexe
7
8 typedef struct FaPont{
9     int pirosak;
10    SzoFa Fiú[26];
11 };
12 FaPont Tar[maxM]; // memória a cellák számára
13 SzoFa UjFapont(){
14     int uj=szabad++;
15     Tar[uj].pirosak=0;
16     for(int i=0;i<26;i++) Tar[uj].Fiú[i]=NIL;
17     return uj;
18 }
```

```
1 void Faba(SzoFa F, string szo){
2     SzoFa p=F;
3     for(int i=0; i<szo.length(); i++){
4         Tar[p].pirosak++;
5         int idx=szo[i]-'a';
6         if (Tar[p].Fiuk[idx]==NIL)
7             Tar[p].Fiuk[idx]=UjFapont(); //uj fapont létrehozása
8         p= Tar[p].Fiuk[idx];
9     }
10    Tar[p].pirosak++;
11 }
12 SzoFa Keres(SzoFa F, string szo){
13     SzoFa p=F;
14     for(int i=0; i<szo.length(); i++){
15         int idx=szo[i]-'a';
16         if (Tar[p].Fiuk[idx]==NIL) return NIL;
17         p= Tar[p].Fiuk[idx];
18     }
19     return p;
20 }
```

```
21 int main (){
22     ios_base::sync_with_stdio(false); cin.tie(NULL);
23     int N,K;
24     string szo;
25     cin>>N>>K;
26     getline(cin,szo);
27     SzoFa F=UjFapont();
28
29     for(int i=0;i<N;i++){
30         getline(cin,szo);
31         Faba(F, szo);
32     }
33
34     for(int k=0;k<K;k++){
35         getline(cin,szo);
36         SzoFa p=Keres(F, szo);
37         if(p!=NIL) cout<<Tar[p].pirosak; else cout<<0;
38         cout<<'\n';
39     }
40 }
```

A Prefixek feladat 2. megoldása

Vegyük észre, hogy elég lenne csak a kérdésekben szereplő szavakat ábrázoló szófát előállítani.



A speciális szavak szófája, a **pirosak** számláló értékekkel.

A **pirosak** értékek úgy is előállíthatók, hogy csak a speciális szavakat tesszük be ebbe a szófába. Majd minden nem speciális szó keresését hajtjuk végre, növeljük az úton érintett pontok **pirosak** értékek, de nem megyünk tovább a ha speciális (kérdésbeli) szót reprezentáló **kék** ponthoz értünk.

A kétirányú lánchoz:

<https://codeforces.com/contest/2012/problem/G>

Szófához:

Hasonlítsuk össze a kétféle megoldás futási idejét!

Oldjuk meg a Prefix feladot a 2. módszer szerint!

A Prefixek feladat megoldását itt lehet értékeltetni:

<https://www.spoj.com/problems/ADAINDEX/>