Statikus adatszerkezetek

Szerző: Horváth Gyula

2025. október 30.

1. 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

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

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

Példa

Bemenet	Kimenet
`aLiLlLuLj`	`julia`
`nevetlekLLLLLBBB`	`szeretlek`
`password1234`	`password1234`

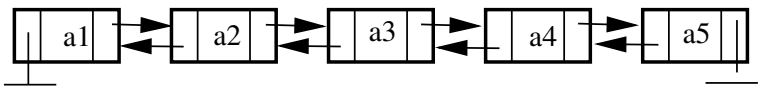
1.1. Megoldás

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 `(a1, a2, a3, a4, a5)` 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:

1.2. statikus ábrázolás

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

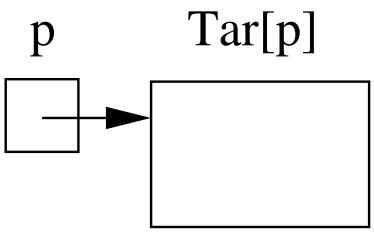
1.3. dinamikus ábrázolás

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

Most a statikus megvalósítást alkalmazzuk.

1.4. Hivatkozás statikus ábrázolás esetén

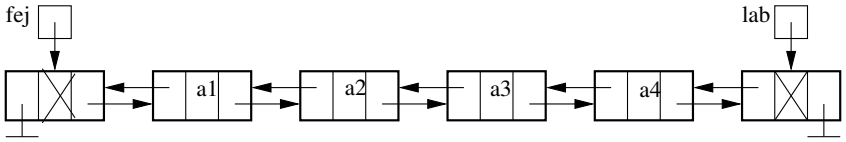
Tegyük fel, hogy a cellák számára *Cella Tar[maxN]* tömbben foglalunk memóriát.
Ekkor a *p* referencia típusú változó, akkor az általa hivatkozott cella a *Tar[p]*



Hivatkozás statikus ábrázolás esetén

2. 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 `(a1, a2, a3, a4)` sorozatot ábrázoló kétirányú lánc **fej** és **lab** cellával.

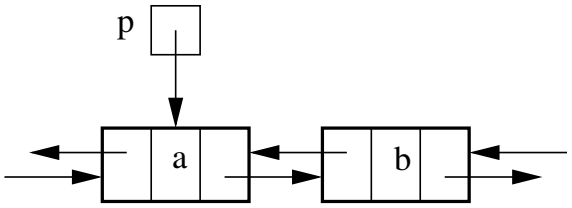
3. A kétirányú lánc statikus adatszerkezet

```
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 eloz, kovet;
8 };
```

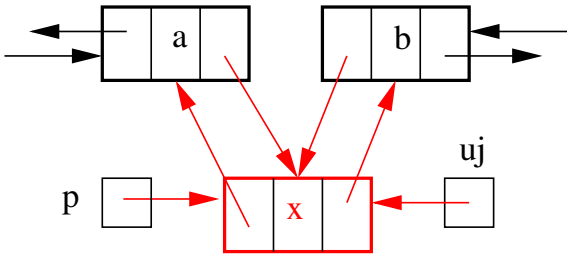
```
1  Cella Tar[maxM];// memória a cellák számára
2  int szabad=1;    //az első szabad cella indexe
3  Ref UjCella(){   //új cella kérése a tárból
4      int idx=szabad;
5      szabad=Tar[szabad].kovet;
6      return idx;
7  }
8  void VisszaAd(Ref c){//a c cella megy a szabadok közé
9      Tar[c].kovet=szabad;
10     szabad=c;
11 }
9  struct Lanc{
10     Ref fej;
11     Ref lab;
12     Lanc(){
13         fej=UjCella(); lab=UjCella();
14         Tar[fej].eloz=NIL; Tar[fej].kovet=lab;
15         Tar[lab].eloz=fej; Tar[lab].kovet=NIL;
16     }
17 };
18 void Kezd(){//a szabad cellák összeláncolása
19     for(int i=1;i<maxM-1;i++)
20         Tar[i].kovet=i+1;
21     Tar[maxM-1].kovet=NIL;
22 }
```

3.1. 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 }
```



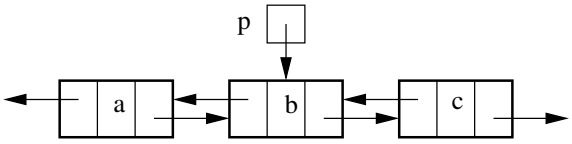
A művelet előtt.



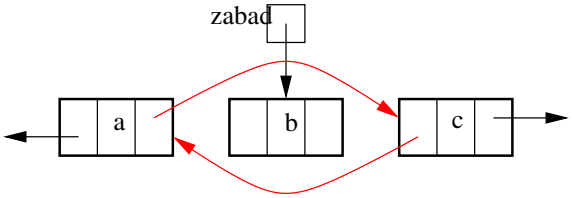
A művelet után.

3.2. 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.

3.3. A sorozat adattípus megvalósítása kétirányú lánccal

```
struct Sorozat{
    Ref fej, lab,  kurzor;
    Sorozat(){//konstruktpr
        fej=UjCella(); lab=UjCella();
        Tar[fej].kovet=lab; Tar[fej].eloz=NIL;
        Tar[lab].kovet=NIL; Tar[lab].eloz=fej;
        kurzor=fej;
    }
// A műveletek deklarációja
    void balra();
    void jobbra();
    void beszur(E x);
    void torol();
    void Felsorol(void muvel(E));
};
```

3.4. A sorozat adattípus műveleteinek megvalósítása

```
    void Sorozat:: balra(){
    if(kurzor!=fej) kurzor=Tar[kurzor].eloz;
    }
    void Sorozat:: jobbra(){
        if(kurzor!=lab) kurzor=Tar[kurzor].kovet;
    }
    void Sorozat:: beszur(E x){
        Lancba(x, kurzor);
    }
    void Sorozat::torol(){
        cellatorol(kurzor);
    }
    void Sorozat:: Felsorol(void muvel(E)){
        Ref p=Tar[fej].kovet;
        while(p!=lab){
            muvel(Tar[p].adat); p=Tar[p].kovet;
        }
    }
```

3.5. A főprogram

```
int main(){
    Sorozat S=Sorozat();//üres sorozet létesítés
    string bemenet;
    getline(cin, bemenet);
    for(int i=0;i<bemenet.size();i++){
        char x=bemenet[i];
        switch(x){
            case 'L': S.balra(); break;
            case 'R': S.jobbra(); break;
            case 'B': S.torol(); break;
            default : S.beszur(x);
        }
    }
    Ref p=Tar[S.fej].kovet;
    while(p!=S.lab){
        cout<<Tar[p].adat;
        p=Tar[p].kovet;
    }cout<<endl;
}
```

3.6. Hatékonyság

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.

```
Ref UjCella(){//új cella kérése a tárból
    int idx=szabad;
    szabad=Tar[szabad].kovet;
    return idx;
}
void VisszaAd(Ref c){//a c cella visszakerül a szabadok közé
    Tar[c].kovet=szabad;
    szabad=c;
}
```

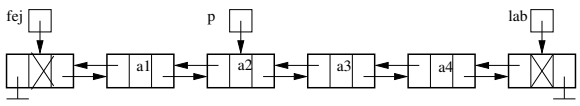
4. További lánc-műveletek

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

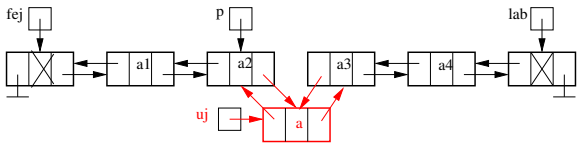
```
1  struct Lanc{
2      Ref fej;
3      Ref lab;
4      Lanc(){
5          fej=UjCella(); lab=UjCella();
6          Tar[fej].eloz=NIL; Tar[fej].kovet=lab;
7          Tar[lab].eloz=fej; Tar[lab].kovet=NIL;
8      }
9  };
10 void Kezd(){//a szabad cellák összeláncolása
11     for(int i=1;i<maxM-1;i++)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

4.1. 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 }
```



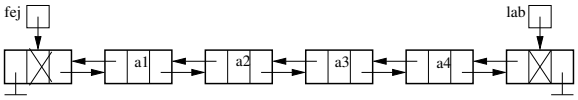
A művelet előtt



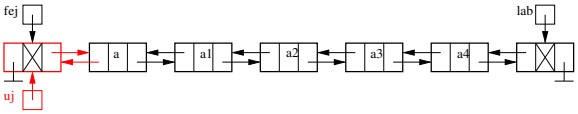
A művelet után

4.2. Elejére művelet

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



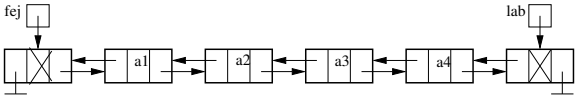
A művelet előtt



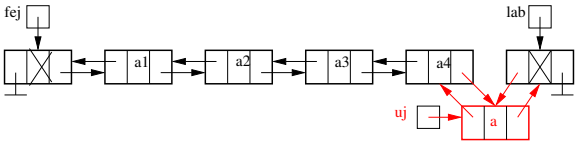
A művelet után

4.3. Végére művelet

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



A művelet előtt

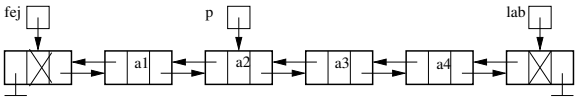


A művelet után

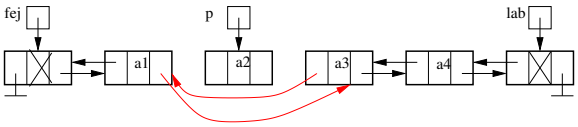
4.4. Töröl művelet

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



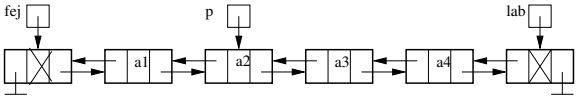
A lánc a művelet után

4.5. Keres és Üresít művelet

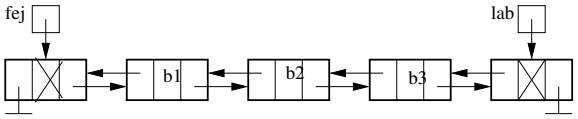
```
1 Ref Keres(Lanc &L, E a){
2     Ref p=L.fej;
3     while(p!=L.lab && a!=Tar[p].adat)
4         p=Tar[p].kovet;
5     if(p==L.lab) return NIL; else return p;
6 }
7 \vspace{-3ex}
8 Uresit(Lanc& L){
9 //ha visszaadjuk a törölt cellákat
10    Ref p=L.fej;
11    while(p!=L.lab){
12        Ref pk=Tar[p].kovet;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

```
15      }
16
17      Tar[L.fej].kovet=L.lab;
18      Tar[L.lab].eloz=L.fej;
19  }
```

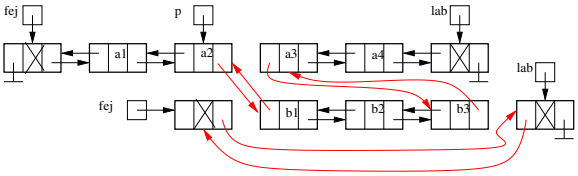
4.6. Láncba láncot beszúr művelet



Az La lánc a művelet előtt



Az Lb lánc a művelet elött



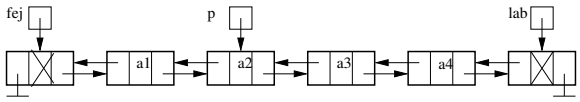
Az La lánc a művelet után. Az Lb lánc ües lesz.

5. Láncba láncot beszúr művelet

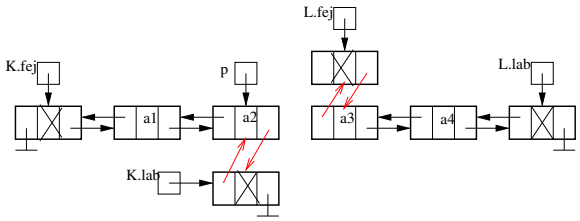
```
1  void LancbaLanc(Lanc& La, Ref p,  Lanc& Lb){
2      if(La.lab==p) return;
3      if(Tar[Lb.fej].kovet==Lb.lab) return;
4      Ref be=Tar[Lb.fej].kovet;
5      Ref bu=Tar[Lb.lab].eloz;
6      Tar[bu].kovet=Tar[p].kovet;
7      Tar[Tar[p].kovet].eloz=bu;
8
9      Tar[p].kovet=be;
10     Tar[be].eloz=p;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

5.1. 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

5.2. Egyesit 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 }
```

5.3. A Felsorol művelet

A **Felsorol** művelet a lánccal 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 }
```

5.4. Index szerinti elérés művelete

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 }
```

5.5. 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 **Uresit** művelet futási ideje minden esetben a lánc hosszával lesz arányos arányos ha visszaadjuk a lánc celláit.
A többi művelet futási ideje konstans.

6. Az adatkezelés szintjei

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

6.1. Absztrakt adattípus

6.2. Adattípus: $A = (E, M)$

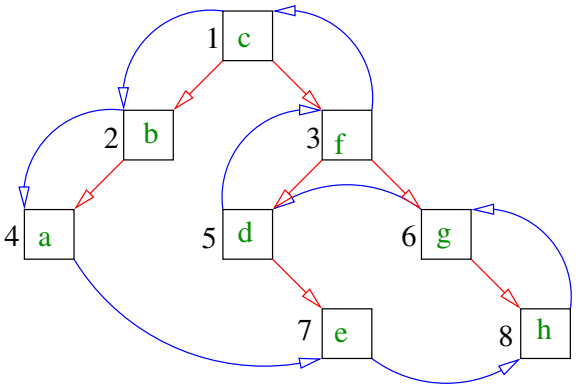
1. E: értékhalmaz,
2. M: műveletek halmaza.

Absztrakt, ha:

i. nem ismert az adatokat tároló adatszerkezet,

ii. nem ismertek a műveleteket megvalósító algoritmusok, a műveletek specifikációjukkal definiáltak.

6.3. Példa absztrakt adatszerkezetre



Adatszerkezet piros és kék szerkezeti kapcsolattal

$M = \{1, 2, 3, 4, 5, 6, 7, 8\}$
kék : $M \to M$
piros : $M \to (M \cup \{\perp\}) \times (M \cup \{\perp\})$
adat : $M \to char$

6.4. 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

6.5. Absztrakt adatszerkezet

6.6. Adatszerkezet: `S = (M, R, Adat)`

1. `M` az absztrakt memóriahelyek, *cellák* halmaza.
2. `R = {r1, ..., rk}` a cellák közötti *szerkezeti kapcsolatok*, `ri : M → (M ∪ {⊥})*`
3. `Adat : M → E` parciális függvény, a cellák adattartalma.

A ⊥ jelentése: nem létező szerkezeti kapcsolat.
`x ∈ M`, `r ∈ R` és `r(x) = ⟨y1, ..., yi, ..., yk⟩`
esetén az `x` cella `r` kapcsolat szerinti szomszédjai `{y1, ..., yk}`,
`yi` pedig az `x` cella `i`-edik szomszédja.
Ha `yi =`⊥, 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.

7. Feladat: várótermi sorbanállás kezelése.

7.1. Probléma szintje

Kezelésre érkező betegek sorbanállását kell kezelni. 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.

7.2. Modell szintje

Minden várakozási sor megadható a matematikai sorozat fogalmát használva.
`Adatsor = (b1, b2, ..., bn)`

7.3. Absztrakt adattípus szintje

Műveleteket definiálunk sorozatokon, amely műveletek a kívánt funkciókat valósítják meg.
Ezen a specifikációját adjuk meg, azt nem, hogy hogyan tároljuk at adatokat (sorozatokat), és milyen algoritmus valósítja meg a műveletet. Pl.
`Vegere(S, b)`: a `b` beteg adatait (TAJ sorszámát) az `S` sorozat végére teszi.

7.4. 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.

7.5. 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.

7.6. Gépi szint

A fordító program megoldja.

8. Feladat: Leggyakoribb kezdőszelet (Prefixek).

Kutatók szövegelemzést végeznek. Összegyűjtöttek 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

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

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

Példa

Bemenet	Kimenet
9 2 fal falu far tekla tele tere teke vele vet fa te	3 4

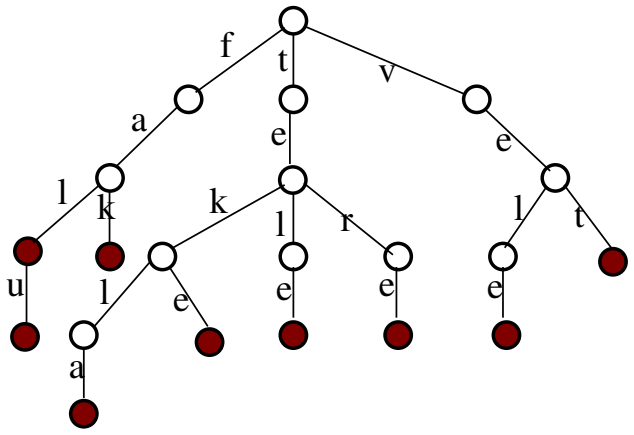
8.1. A naiv megoldás hatékonysága

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.

9. 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.

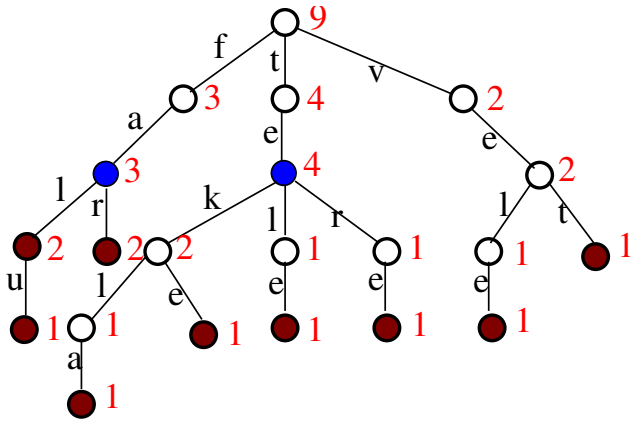
9.1. A példa bemenetet reprezentáló szófa



A piros pontokig vezető utak alkotják a reprezentálandó halmaz elemeit.

9.2. 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.

10. 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.

10.1. 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
```

```
typedef struct FaPont{
    int van;
    SzoFa Fiuk[26];
    FaPont(){
        van=false;
      for (int i=0;i<26;i++) Fiuk[i]=NIL;
    }
};
FaPont Tar[maxM];// memória a cellák számára
```

10.2. Szófa alapművelet: Faba

```
SzoFa UjFapont(){
   int uj=szabad++;
   Tar[uj].van=false;
   for(int i=0;i<26;i++) Tar[uj].Fiuk[i]=NIL;
   return uj;
}
void Faba(SzoFa& F, string szo){
   SzoFa p=F;
   for(int i=0; i<szo.length(); i++){
      int idx=szo[i]-'a';
      if (Tar[p].Fiuk[idx]==NIL){//uj fapont kérése
         Tar[p].Fiuk[idx]=UjFapont();
      }
      p=Tar[p].Fiuk[idx];//tovább az i-edik fiú felé
   }
   Tar[p].van=true; //sz bekerült a halmazba
}
```

11. Szófa alapműveletek:Keres, Torol

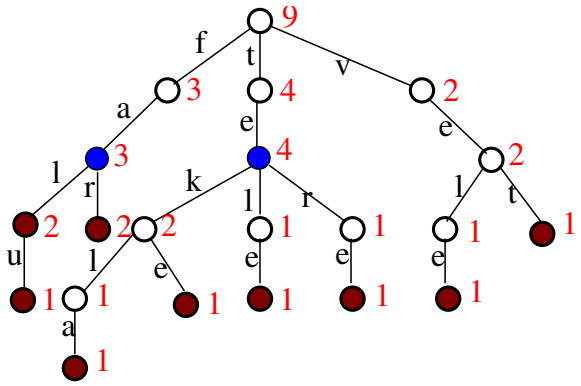
```
1  SzoFa Keres(SzoFa& F, string szo){
2      SzoFa p=F;
3      for(int i=0; i<szo.length(); i++){
4          int idx=szo[i]-'a';
5          if(Tar[p].Fiuk[idx]==NIL) return NIL;
6          p=Tar[p].Fiuk[idx];
7      }
8      return p;
9  }
10 bool Eleme(Szofa& F, string szo){
11   Szofa p=Keres(F,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

11.1. 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  }
```

12. 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.

12.1. A Prefixek feladat 1. megoldása

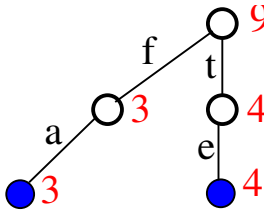
```
1  struct FaPont;
2  typedef int SzoFa;//tömbelem indexe
3  #define NIL -1     //nemlétező cella referenciája
4  typedef char E;  //a sorozet elemeinek a típusa
5  const int maxM=1000001;
6  int szabad=1;    //az első szabad cella indexe
7
8  typedef struct FaPont{
9      int pirosak;
10     SzoFa Fiu[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].Fiu[i]=NIL;
17    return uj;
18 }
```

```
12 void Faba(SzoFa F, string szo){
13     SzoFa p=F;
14     for(int i=0; i<szo.length(); i++){
15         Tar[p].pirosak++;
16         int idx=szo[i]-'a';
17         if (Tar[p].Fiuk[idx]==NIL)
18             Tar[p].Fiuk[idx]=UjFapont();//uj fapont létesítése
19         p=Tar[p].Fiuk[idx];
20     }
21     Tar[p].pirosak++;
22 }
23 SzoFa Keres(SzoFa F, string szo){
24     SzoFa p=F;
25     for(int i=0; i<szo.length(); i++){
26         int idx=szo[i]-'a';
27         if(Tar[p].Fiuk[idx]==NIL) return NIL;
28         p=Tar[p].Fiuk[idx];
29     }
30     return p;
31 }
```

```
32 int main (){
33     ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL)
34     int N,K;
35     string szo;
36     cin>>N>>K;
37     getline(cin,szo);
38     SzoFa F=UjFapont();
39 
40     for(int i=0;i<N;i++){
41         getline(cin,szo);
42         Faba(F, szo);
43     }
44 
45     for(int k=0;k<K;k++){
46         getline(cin,szo);
47         SzoFa p=Keres(F, szo);
48       if(p!=NIL) cout<<Tar[p].pirosak; else cout<<0;
49       cout<<''\n'';
50     }
51 }
```

13. 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.

14. Gyakorló feladatok

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/