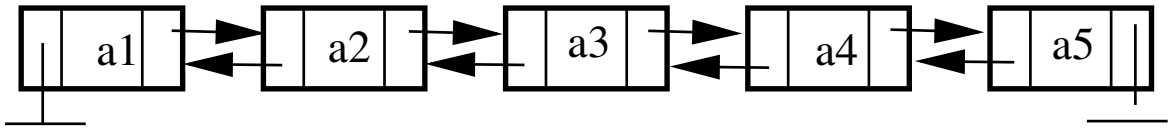
Dinamikus adatszerkezetek

Szerző: Horváth Gyula
Előadó: Busa Máté

2025. szeptember 11.

Az adatkezelés szintjei

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



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:

statikus

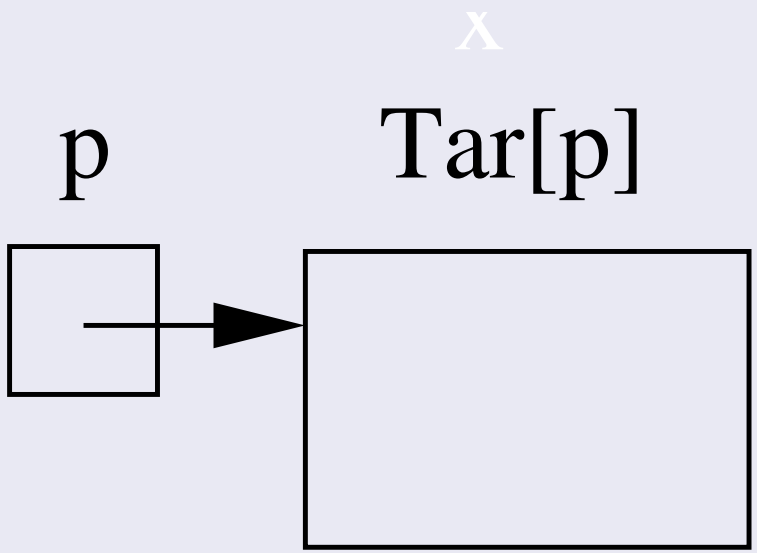
1. Minden cella számára egy tömb, vagy dinamikus tömb elemeként foglalunk memória helyet.
2. A referencia ekkor a cellát tartalmazó tömbelem indexe.

dinamikus

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

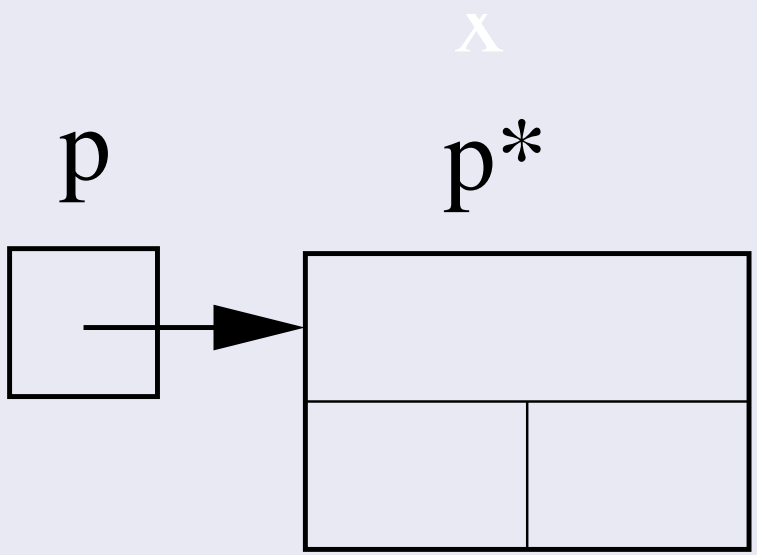
Hivatkozás statikus ábrázolás esetén

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]



Hivatkozás dinamikus ábrázolás esetén

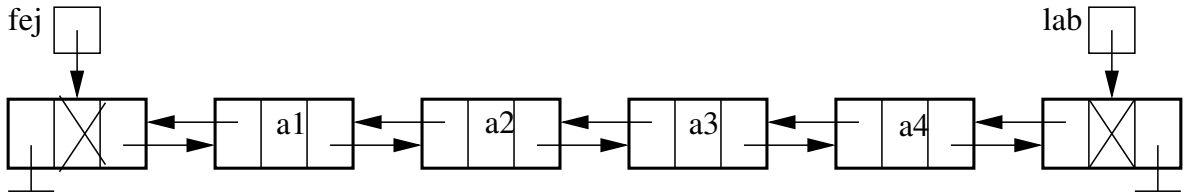
Tegyük fel, hogy a cellák számára dinamikusa, a new Cella() művelettel foglalunk memóriát.
Ekkor a p referencia típusú változó, akkor az általa hivatkozott cella a p*
Ha p* típusa helyett a olyan struct, amelynek adattagja tag, akkor p*.tag helyett helyett a p->tag is használható.



Dereferencia dinamikus változókra.

Sorozatok ábrázolása kétirányú lánccal

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.

A kétirányú lánc dinamikus adatszerkezet

```
1  struct Cella;//előre deklarálás
2  #define NIL nullptr
3  typedef int E;//a sorozet elemeinek a típusa
4  typedef Cella* Ref;
5  struct Cella{
6      E adat;
7      Ref eloz,kovet;
8      Cella(E a, Ref ecsat, Ref kcsat){//konstruktor
9          adat=a; eloz=ecsat; kovet=kcsat;
10     }
11     Cella(){//paramétermentes konstruktor
12         eloz=NIL; kovet=NIL;
13     }
14 };
```

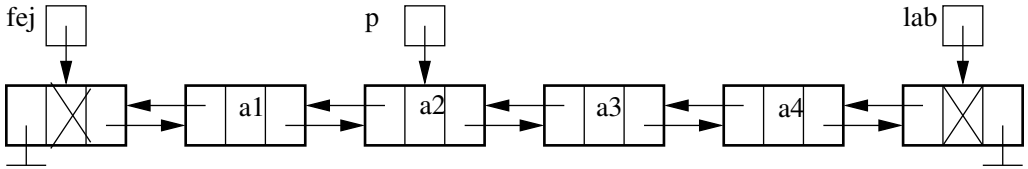
A Lanc típus

```
1  struct Lanc{
2      Ref fej;
3      Ref lab;
4      Lanc(){//konstruktor
5          fej=new Cella(); lab=new Cella();
6          fej->kovet=lab;
7          lab->eloz=fej;
8      }
9  };
```

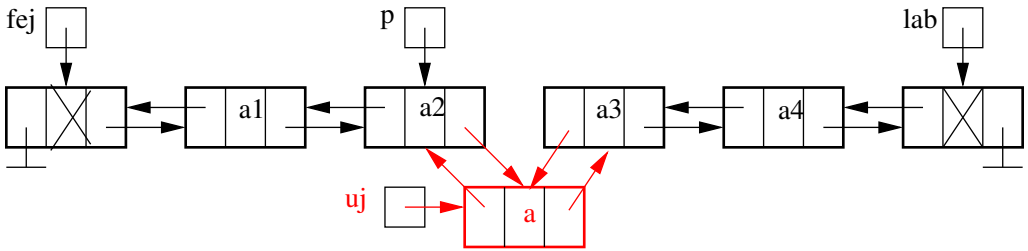
Lancba művelet

Új cella beszúrása a p cella után az a adattartalommal.

```
1  Ref Lancba(Lanc &L, Ref p, E a){
2      if(p==L.lab) return NIL;
3      Ref pkov=p->kovet;
4      Ref uj=new Cella(a, p, pkov);
5      Ref kov=p->kovet;
6      p->kovet=uj;
7      pkov->eloz=uj;
8      return uj;
9  }
```



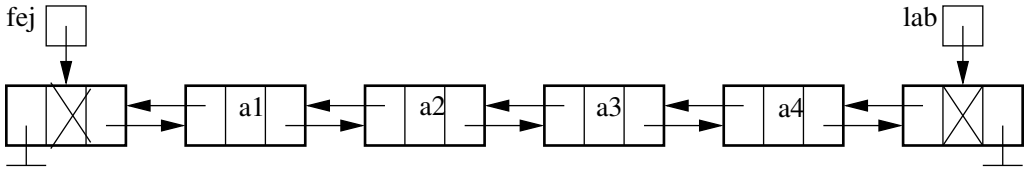
A művelet előtt



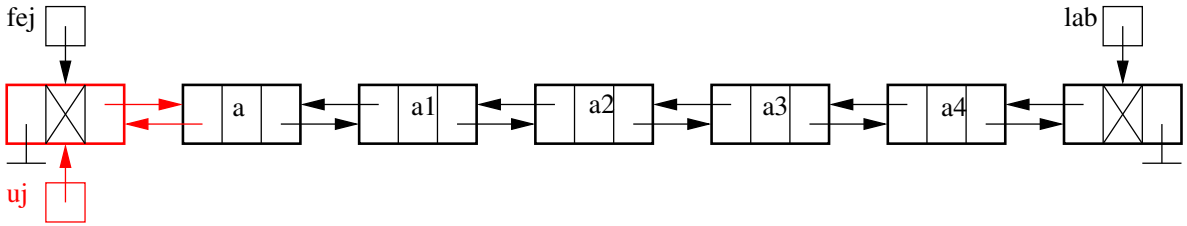
A művelet után

Elejére művelet

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



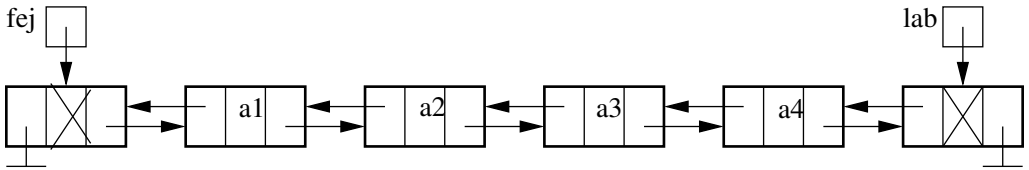
A művelet előtt



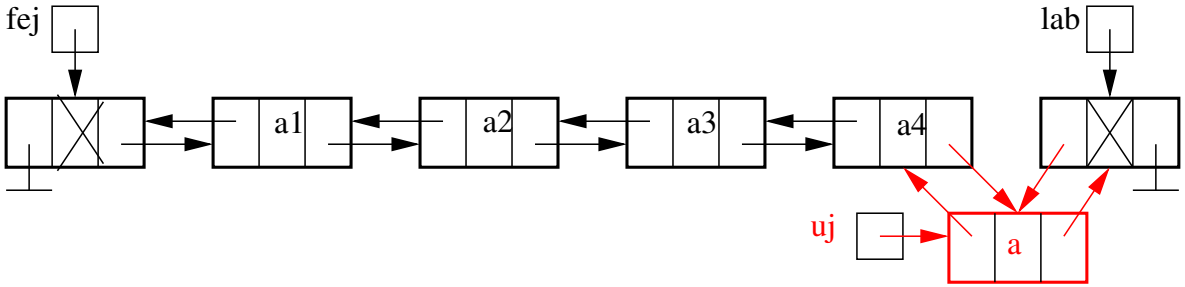
A művelet után

Végére művelet

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



A művelet előtt

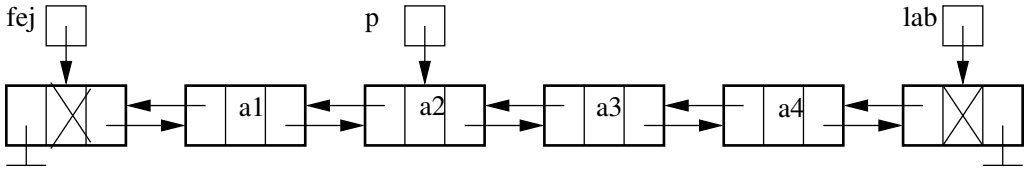


A művelet után

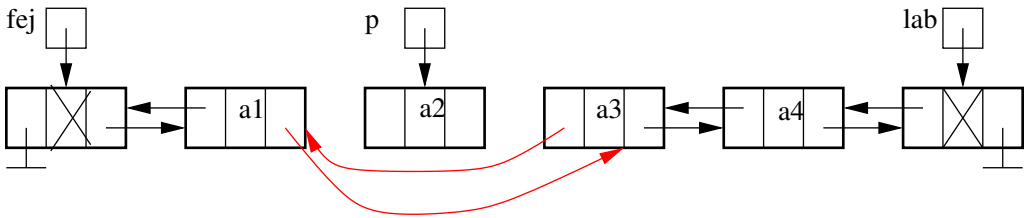
Töröl művelet

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=p->eloz;
4      Ref u=p->kovet;
5      e->kovet=u;
6      u->eloz=e;
7      return u;
8  }
```



A lánc a művelet előtt

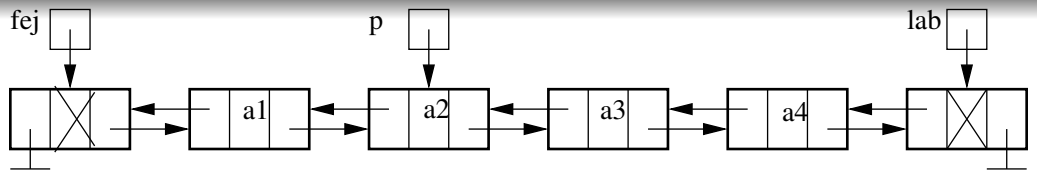


A lánc a művelet után

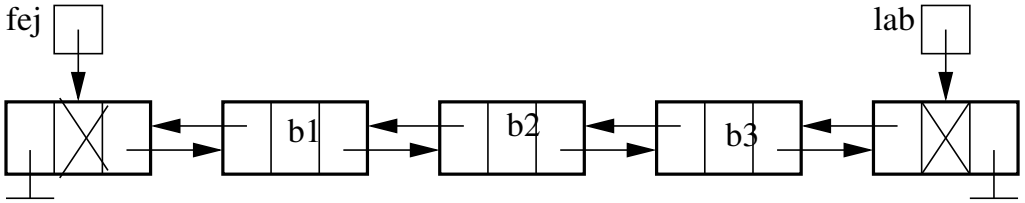
Keres és Üresít művelet

```
1  Ref Keres(Lanc &L, E a){
2      Ref p=L.fej;
3      while(p!=L.lab && a!=p->adat)
4          p=p->kovet;
5      if(p==L.lab) return NIL; else return p;
6  }
7
8  Uresit(Lanc& L){
9  //ha biztosírunk újrahasznosítást
10     L.fej->kovet=L.lab;
11     L.lab->eloz=L.fej;
12 }
```

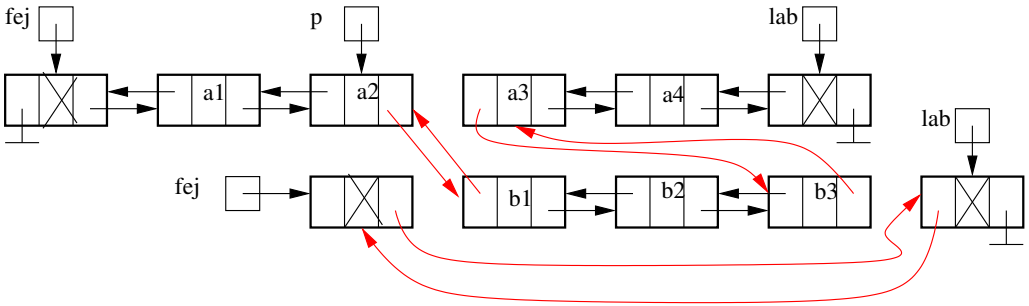
Láncba láncot beszúr művelet



Az *La* lánc a művelet előtt



Az *Lb* lánc a művelet elött



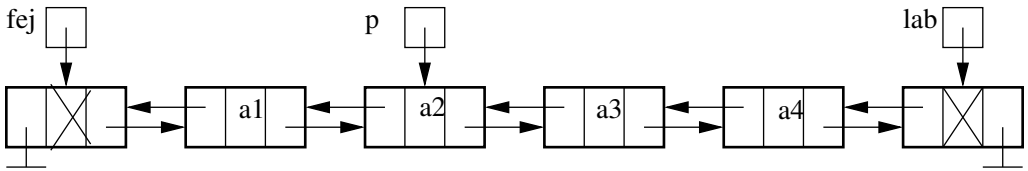
Az *La* lánc a művelet után. Az *Lb* lánc ües lesz.

Láncba láncot beszúr művelet

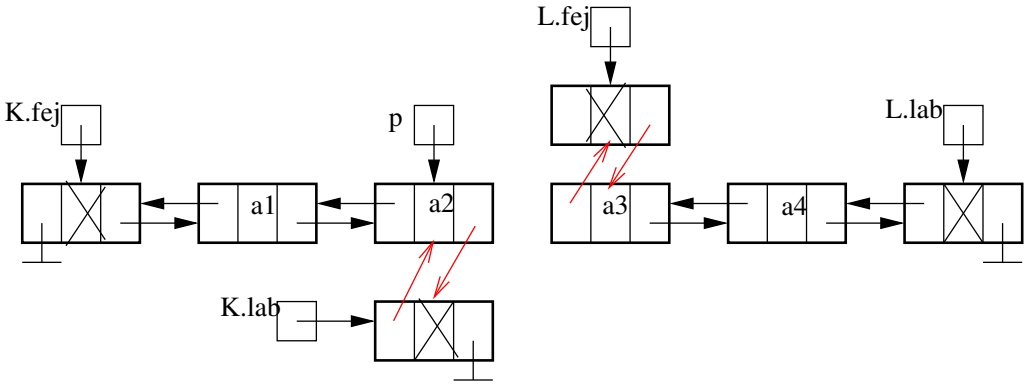
```
1  void LancbaLanc(Lanc& La, Ref p,  Lanc& Lb){
2      if(La.lab==p) return;
3      if(La.fej->kovet==Lb.lab) return;
4      Ref be=Lb.fej->kovet;
5      Ref bu=Lb.lab->eloz;
6      bu->kovet=p->kovet;
7      p->kovet->eloz=bu;
8
9      p->kovet=be;
10     be->eloz=p;
11     Lb.fej->kovet=Lb.lab;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



A K lánc a művelet előtt



A K és L lánc a művelet után

Egyesit és Klon műveletek

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

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=L.fej->kovet;
3      while(c!=L.lab){
4          Muvel(c->adat);
5          c=c->kovet;
6      }
7  }
```

Index szerinti elérés művelete

Az első elem indexe 0.

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

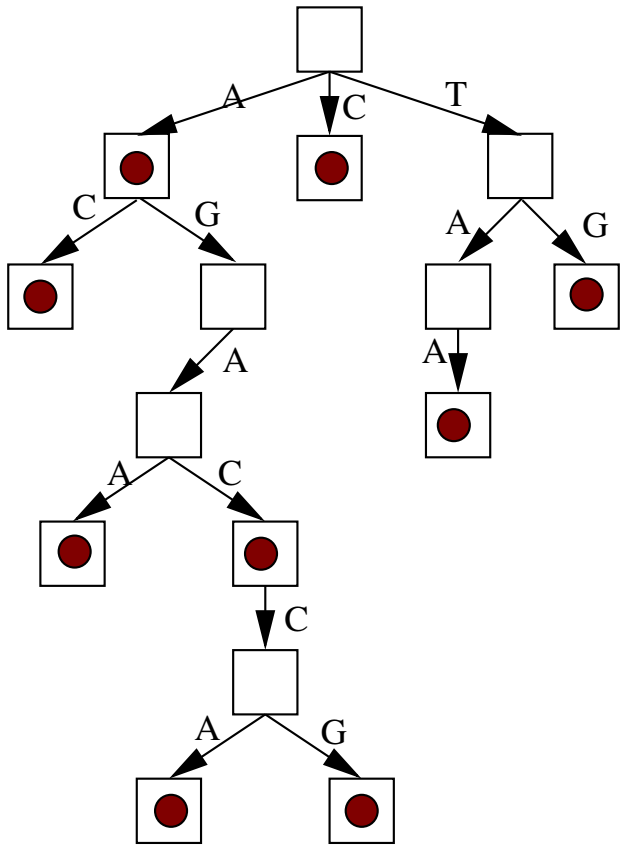
Feladat: DNS szekvenciák vizsgálata

Kutatók DNS szekvenciákat vizsgálnak. Összegyűjtöttek N szekvenciát. Azt akarják megtudni, hogy melyik az a vizsgált szekvencia, amelynek a legtöbb kezdőszelete is ott van a vizsgáltak között. Ha több ilyen van, akkor a lexikografikusan legkisebbet kell megadni. Ha nincs ilyen, akkor az üres sort kell kiírni!

Példa

Bemenet	Kimenet
9 AC C A AGAA AGAC AGACCA AGACCG TAA TG	AGACCA

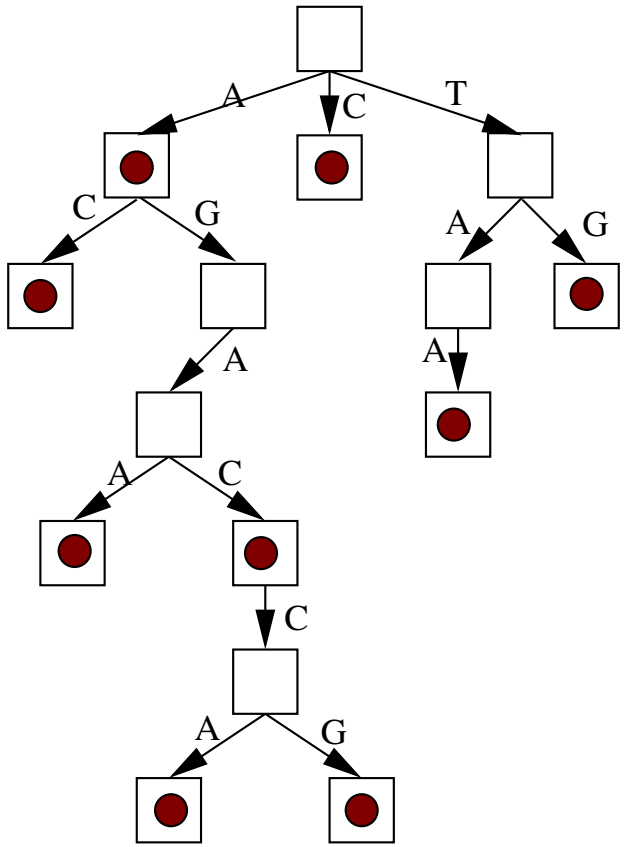
A szekvenciák, mint stringek halmazát ábrázoljuk olyan fában„ amelyre teljesül, hogy minden elágazás a szekvenciákban előforduló karakterek egyikével van címkézve. A fára teljesül, hogy minden S szekvenciához van olyan p pontja a fának, hogy a gyökértől a p-ig vezető úton lévő címkék sorozata megegyezik S-el. és ez a fa p pontjában jelezve van.



A példa bemenetet ábrázoló szófa

A feladat megoldás

A fának ahhoz a p pontjához vezető címke sorozat a megoldás szekvencia, amely ponthoz vezető úton a legtöbb megjelölt pont van, tehát amely pontok olyan szekvenciát reprezentálnak, szintén elemi a bemenetnek.
Példánkban ez a AGACCA szekvencia.



Adatszerkezet választás

Eddig a Szófa absztrakt adatszerkezet szintjén lett megfogalmazva, még nem mondtuk meg, hogy hogyan foglalunk memóriát a fa pontjainak, és hogyan adjuk meg a szerkezeti kapcsolatokat, azaz a fában az elágazásokat.
A megalapozott döntéshez figyelembe kell venni, hogy milyen műveleteket kell végezni az megoldás algoritmusában.
Mindenképpen kell egy olyan művelet - **Faba(f,szo)** -, amely beilleszteni az **f** fába a **szo** szekvenciát.
Fabejárással meg tudjuk határozni a fa minden p pontjára, hogy a gyökértől p-ig vezető úton hány megjelölt pont van, tehát hány kezdőszelet szerepel a bemenetben.

Megoldási módszerek

Meg kell tudni adni, hogy ha a p pont a megoldás, akkor hogyan adjuk meg megadni azt a megoldás szekvenciát, amelyet a p pont reprezentál, mert ki kell íratni.
Több lehetőség közül választhatunk:

1. Tároljuk a bemenetben megadott szekvenciákat egy DNS tömbben és a szavak tömb-indexét tároljuk a fa megfelelő pontjában. A szófa minden p pontban a van adattag értéke a beszúrandó szó DNS tömbbeli indexe, ha a p pont reprezentálja a beszúrandó szót, egyébként pedig -1.
   A szófa bejárásával minden pontra meghatározzuk, hogy hány szavat tartalmaz az addig vezető út, és megjegyezzük a legjobbikat.
2. Minden pontban tároljuk a közvetlen közvetlen ősére mutató referenciát és azt, hogy a pont melyik karakter szerinti fia az apjának, így visszafelé haladva ki tudjuk íratni a megoldás. Bejárással lehet meghatározni a megoldás pontot.

Szófa dinamikus adatszerkezet az 1. módszerhez

```
1  #define NIL nullptr
2  struct FaPont;
3  typedef FaPont* Ref;
4
5  typedef struct FaPont{
6       int van;    //a szó indexe, vagy -1
7       Ref Fiuk[4];
8       FaPont(int idx){
9         van=idx;
10        for (int i=0;i<4;i++) Fiuk[i]=NIL;
11      }
12 };
13 struct SzoFa{
14     Ref gyoker;
15     SzoFa(){
16        gyoker=new FaPont(-1);
17     }
18 };
```

Az 1. módszer algoritmusa

Kerakterhez tömb index rendelés

```
1  int Sx(char x){//karakterek Fiuk-beli indexei
2    switch(x){
3      case 'A': return 0;
4      case 'C': return 1;
5      case 'G': return 2;
6      case 'T': return 3;
7    }
8  }
```

```
 9  vector<string> DNS;//a feladat bemenete
10
11  void Faba(SzoFa F, int idx){
12  //A szekvenciákat tároló DNS dtömb globális
13  //A DNS[idx]  szót kell beilleszteni a szófába
14      Ref p=F.gyoker;
15      for(int i=0; i<DNS[idx].size(); i++){
16          char x=DNS[idx][i];
17          if (p->Fiuk[Sx(x)]==NIL){
18              Ref uj=new FaPont(-1);
19              p->Fiuk[Sx(x)]=uj;
20          }
21          p=p->Fiuk[Sx(x)];
22      }
23      p->van=idx;
24  }
```

```
25  int megoidx=0;//a megoldas prefix sorszáma
26  int maxm=0;    //a leghosstabb prefix hossza
27
28  void Bejar(Ref p, int mely){
29      if(p->van>=0) mely++;
30      if(p->van>=0 && mely>maxm){//hosszabb prefixet t
31          maxm=mely; megoidx=p->van;//bejegyezzük
32      }
33      for(int i=0;i<4;i++)
34          if(p->Fiuk[i]!=NIL)
35              Bejar(p->Fiuk[i], mely);
36  }
```

```
37  int main (){
38      ios_base::sync_with_stdio(false); cin.tie(NULL);
39      int n,m;
40      string s;
41      cin>>n;
42
43      getline(cin,s);
44      SzoFa f;
45      DNS.push_back("");
46      for(int i=1;i<=n;i++){
47          getline(cin,s);
48          DNS.push_back(s);
49          Faba(f, i);
50      }
51      //minden szóra kiszámítja, hogy hány prefixe van
52      //megoidx a leghosszabb DNS-beli indexe lesz les
53      Bejar(f.gyoker,-1);
54
55      cout<<DNS[megoidx]<<endl;
56  }
```

Szófa dinamikus adatszerkezet az 2. módszerhez

```
1  define NIL nullptr
2  struct FaPont;
3  typedef FaPont* Ref;
4
5  typedef struct FaPont{
6      bool van;
7      Ref Fiuk[4];
8      pair<Ref, char> Apa;
9      FaPont(Ref os, char fiu){
10         van=false;
11         for (int i=0;i<4;i++) Fiuk[i]=NIL;
12         Apa.first=os; Apa.second=fiu;
13     }
14 };
15 struct SzoFa{
16     Ref gyoker;
17     SzoFa(){ gyoker=new FaPont(NIL, '0');
18     }
19 };
```

Az 2. módszer algoritmusa

```
1  int c2i[256];//karakterek Fiu-beli indexei
2
3  void Faba(SzoFa& F, string szo){
4      Ref p=F.gyoker;
5      for(int i=0; i<szo.size();i++){
6          int fi=c2i[szo[i]];
7          if (p->Fiuk[fi]==NIL){
8              p->Fiuk[fi]=new FaPont(p, szo[i]);
9          }
10         p=p->Fiuk[fi];
11     }
12     p->van=true;
13 }
```

```
1  int maxp=0;          //leghosszabb prefix hosssza
2  Ref maxpont=NIL;//a leghosszabb prefix végpontja
3
4  void Bejar(Ref f, int mely){
5      Ref p;
6      if(f->van && mely>maxp){//hosszabb prefixet talá
7          maxp=mely; maxpont=f;//megjegyezzük a végpont
8      }
9      for(int i=0; i<4;i++){
10         p=f->Fiuk[i];
11         if(p==NIL) continue;
12         if (p->van)
13             Bejar(p, mely+1);
14         else
15             Bejar(p, mely);
16      }
17 }
```

```
1   ios_base::sync_with_stdio(false); cin.tie(NULL);
2   int n;
3   string szo;
4   c2i['A']=0; c2i['C']=1; c2i['G']=2; c2i['T']=3;
5   cin>>n;
6   getline(cin,szo);
7   SzoFa F=SzoFa();
8
9   for(int i=0;i<n;i++){//Szófa felépítése
10      getline(cin,szo);
11      Faba(F, szo);
12  }
```

```
13      //minden szóra kiszámítja, hogy hány prefixe van
14      //maxpont a leghosszabb végpontja lesz
15      Bejar(F.gyoker, 0);
16
17      vector<char> prefix;
18       //a leghosszabb prefix előállítása fordított so
19      while(maxpont!=F.gyoker){
20          auto os=maxpont->Apa;
21          prefix.push_back(os.second);
22          maxpont=os.first;
23      }
24      reverse(prefix.begin(), prefix.end());//megfordí
25      for(char x:prefix) cout<<x; cout<<endl;
```

További Szófa műveletek

Látható, hogy string elemeket tartalmazó halmazok alapvető műveletei: Faba, Kivesz, Keres hatékonyan megvalósíthatók Szófával; mindegyik futási ideje a string hosszával arányos lesz.

```
1  Ref Keres(SzoFa f, string szo){
2      Ref p=f.gyoker;
3      for(int i=0;i<szo.size();i++){
4          if(p->Fiuk[c2i(s[i])]==NIL) return NIL;
5          if(p->van>=0) return p;
6          p=p->Fiuk[c2i(s[i])];
7      }
8  }
9
10 void Kivesz(SzoFa f, string s){
11     Ref p=Keres(f, s);
12     if(p!=NIL)
13         p->van=-1;
14 }
```

Hatékonyság: memória igény

A memória igény a szófa pontjainak a számával arányos. Legyen Px azon j indexek halmaza, amely indexű bemeneti szó kezdőszelete valamely másik szónak. Ekkor a szófa pontjainak száma

$$nP = \sum_{i=1}^{N} szo\,[\,i\,]\,.size - \sum_{j \in Px} szo\,[\,j\,]\,.size$$

A tényleges memória igényt befolyásolja a Fiuk[] tömb számára foglalandó memória, azaz a fiók lehetséges száma. Ha K a lehetséges karakterek száma és I minden lehetséges karakter számára foglalunk helyet a Fiuk[] tömbben, akkor a szófa ábrázolásához kellő memória: $nP * K * (4 + e)$, ahol e az egyéb adatok igénye.

Memória igény csökkentése

Ha a szavak tetszőleges karaktereket tartalmazhatnak, akkor nagyon jelentős lehet a memória igény. Még akkor is ha csak az (angol) alfabetikus karakterek lehetnek a szavakban.
A memóriaigény csökkenthető esetlegesen a futási idő növekedése kárára.
Ez többféleképpen is elérhető. Az alapelv az, hogy miden fapontban csak azon karakterek szerinti elágazásoknak foglalunk helyet, amelyek ténylegesen kellenek.
Tehát minden olyan módszer alkalmazható, amely általános fák dinamikus ábrázolására alkalmas.
A legegyszerűbb megoldás az, ha a Ref Fiuk[] tömb helyett dinamikus tömböt, vektort használunk: vector<pair<char,Ref» Fiuk. Ekkor persze lineáris kereséssel kell elérni az adott karakterhez tartozó elágazást, ha nincs, akkor a végére veszünk fel.
A fiuk sorozatát láncban is lehet tárolni.

Szófa ábrázolás: Fiuk dinamikus tömbben

```
1  struct FaPont;
2  typedef FaPont* SzoFa;
3  #define NIL nullptr //nemlétező cella referenciája
4  typedef char E;  //a sorozet elemeinek a típusa
5
6  typedef struct FaPont{
7       bool van;
8       vector<pair<char,SzoFa>> Fiuk;
9       FaPont(){
10          van=false;
11      }
12 };
```

```
1  void Faba(SzoFa F, string szo){
2      SzoFa p=F;
3      FaPont ujfiu;
4      for(int i=0; i<szo.length(); i++){
5          char c=szo[i]; int cx=0;
6          if(p->Fiuk.size()==0){
7              p->Fiuk.push_back({c,NIL});
8          }else{
9             while(cx<p->Fiuk.size()&&
10                 p->Fiuk[cx].first!=c)
11                     cx++;
12             if(cx==p->Fiuk.size())
13                 p->Fiuk.push_back({c,NIL});
14         }
15         if (p->Fiuk[cx].second==NIL){
16             p->Fiuk[cx].second=new FaPont();
17         }
18         p=p->Fiuk[cx].second;
19     }
20     p->van=true;
21 }
```

```
22  SzoFa Keres(SzoFa F, string szo){
23      SzoFa p=F;
24      for(int i=0; i<szo.length(); i++){
25          char c=szo[i];
26          int cx=0;
27          while(cx<p->Fiuk.size() &&
28                p->Fiuk[cx].first!=c) cx++;
29
30          if(cx==p->Fiuk.size() ||
31                p->Fiuk[cx].first!=c) return NIL;
32          p=p->Fiuk[cx].second;
33      }
34      return p;
35  }
```

```
36  bool Torol(SzoFa F, string s){
37      SzoFa p=Keres(F, s);
38      if(p!=NIL){
39          p->van=false;
40          return true;
41      }
42      return false;
43  }
```

Megjegyezzük, hogy ebben a reprezentóban nem okoz gondot az ékezetes karakterek használata.

Gyakorló feladatok

1. https://mester.inf.elte.hu: Rekurzív adatszerkezetek/Legtöbbször előforduló kezdőszelet
2. https://dmoj.ca/problem/ioi08p1 (IOI2008 Printer feladat)
3. https://open.kattis.com/problems/babynames
4. Oldjuk meg a prefixek feladatot úgy, hogy a kérdéseket tároljuk szófában és az elágazásokat dinamikus többen tároljuk: https://www.spoj.com/problems/ADAINDEX/