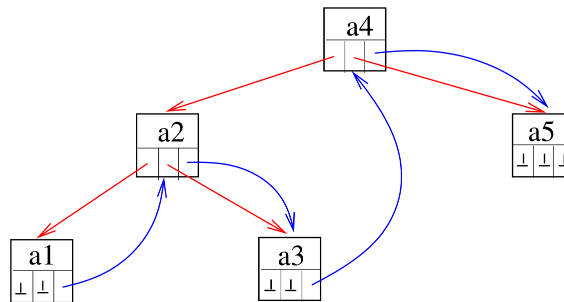
# 5. Adatszerkezetek

Egy $\mathtt{A} = (M, R, Adat)$ absztrakt adatszerkezet megvalósítása:
1. Konkrét memória allokálás az $M$-beli absztrakt memória cellák számára.
2. Az $R$ szerkezeti kapcsolatok ábrázolása.
3. Alapműveletek algoritmusainak megadása.

**Belső adatszerkezet**
A cellákat a főtárban lefoglalt memóriamezők tárolják. Minden cellát a számara lefoglalt memóriamező kezdőcíme azonosít.

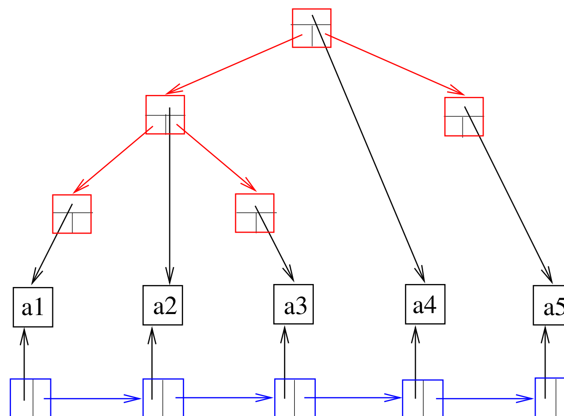
**Külső adatszerkezet**
A cellákat külső tárolón (lemez) fájl tárolja. Minden cellát egy $(F, p)$ pár azonosít, ahol $F$ a tároló fájl azonosítója és $p$ az $F$ fájlon belüli rekordsorszám.

**Elosztott adatszerkezet**
Az egyes cellákat különböző számítógépeken tarolhatjuk. Egy cella azonosításához egy $(G, F, p)$ hármast kell megadni, ahol $G$ a hálózatba kapcsolt számítógép (adott hálózati protokoll szerinti) azonosítója, $F$ a fájl azonosítója és $p$ a fájlon belüli rekordsorszám.
**Endogén ábrázolás**
Az adatot és a szerkezeti kapcsolatot ugyanaz a cella tartalmazza.



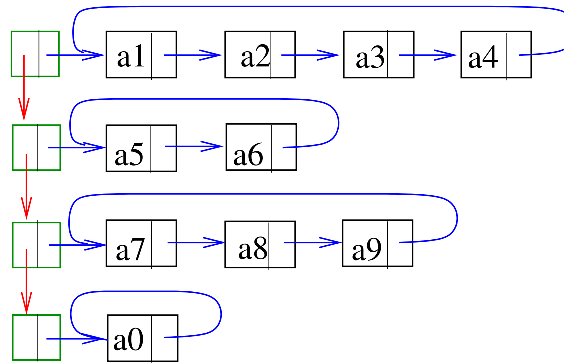

1. ábra. Adatszerkezet endogén ábrázolása.

**Exogén ábrázolás**
Külön cella tartalmazza az adatot és a szerkezeti kapcsolatokat. A szerkezeti kapcsolatot tartalmazó cellában a megfelelő adatra mutató hivatkozást tároljuk.
Objektum orientált programozási nyelv esetén az adatszerkezetek ábrázolása alapvetően exogén. Például a java esetén csak akkor alkalmazható endogén ábrázolás, ha az adatok típusa elemi típus (int, long, float, double, char, boolean).



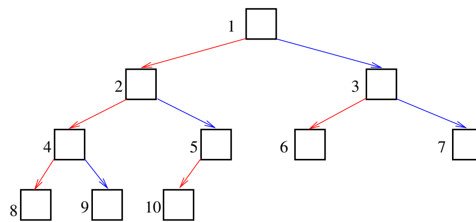

2. ábra. Adatszerkezet exogén ábrázolása.

**Heterogén ábrázolás**
Egyes cellák csak szerkezeti kapcsolatot tartalmaznak, mások tartalmazhatnak adatot és a szerkezeti kapcsolatokat.




3. ábra. Adatszerkezet heterogén ábrázolása.

**Statikus ábrázolás**
A memória lefoglalás statikus tömbbel történik. Minden cellát tömbbeli indexe azonosít. Hiányzó szomszédot a 0 (vagy -1) index azonosítja. **Statikus ábrázolás megvalósítása:**




4. ábra. Minta fa.

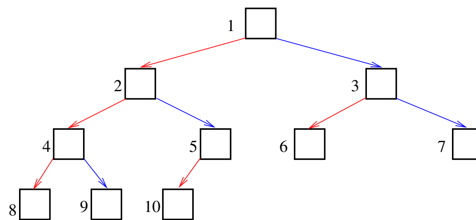
Az algoritmus során az adatszerkezet új cellával bővülhet, illetve cellát törölhetünk is. Ezért mindig tudnunk kell, hogy a tömb mely elemei szabadok, azaz melyek használhatók bővítéskor. Ez megoldható úgy, hogy az egyik szerkezeti kapcsolatot, esetünkben a bal fiút arra használjuk, hogy a szabad cellákat egy láncba gyűjtsük. a *Szabad* mező tartalmazza mindig a lánc fejét.

```
public class BinFaS<E>{
   private static class Cella<E>{
      E elem;
      int bal, jobb;
   }
   public Cella<E>[] Tar;
   public int gyoker;
   public int szabad;

   BinFaS(int maxn){
      Tar=(Cella<E>[])new Cella[maxn];
      gyoker=-1;
      szabad=0;
      for (int i=0; i<maxn-1; i++)
         Tar[i].bal=i+1;
      Tar[maxn-1].bal=-1;
   }
   public int UjPont(){
      if (szabad==-1)
         return -1;
```

```
      int p=szabad;
      szabad=Tar[szabad].bal;
      return p;
   }
   public void PontTorol(int p){
      Tar[p].bal=szabad;
      szabad=p;
   }
}
```

**Dinamikus ábrázolás**
A cellák számára dinamikusan foglalunk memóriát, minden cellát pointer érték - memóriacím - azonosít. A hiányzó kapcsolatot a *null* (*nil*) pointer érték ábrázolja. A szerkezeti kapcsolatot, mint pointer értéket a cellában tároljuk.
**Szerkezeti kapcsolat számítása**
A szerkezeti kapcsolat esetenként megadható számítási eljárással is, nincs szükség a szerkezeti kapcsolat tárolására. Minden $x$



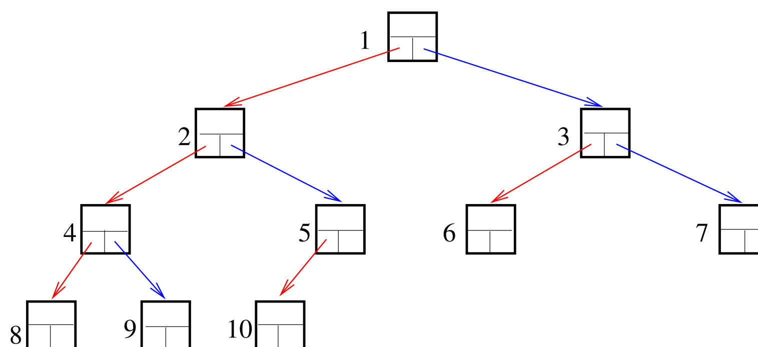

5. ábra. Szerkezeti kapcsolat megadása számítási eljárással.

cellára és $r$ szerkezeti kapcsolatra és adott $i$-re kiszámítható $x$-nek $r$-szerinti $i$-edik szomszédja. Például, az alábbi fa esetén, ha $M = \{1, \dots, n\}$, és az $R = \{bal, jobb\}$ szerkezeti kapcsolat olyan, hogy

$$bal(i) = \begin{cases} 2i & \text{ha } 2i \leq n \\ \perp & \text{ha } 2i > n \end{cases}$$

$$jobb(i) = \begin{cases} 2i+1 & \text{ha } 2i+1 \leq n \\ \perp & \text{ha } 2i+1 > n. \end{cases}$$

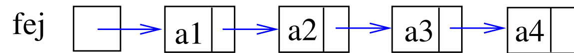
A cellák számára statikus tömbbel foglalhatunk memóriát. **Szerkezeti kapcsolat tárolása**
A fenti fát megvalósíthatjuk a szerkezeti kapcsolatok tárolásával is. Ekkor minden cella tartalmazza a szerkezeti kapcsolat szerinti szomszédait (és az adatot).




6. ábra. Szerkezeti kapcsolat tárolása.

## 5.1. Lánc

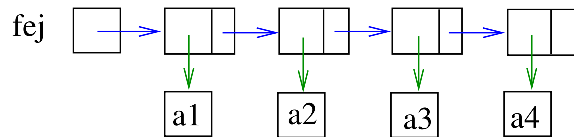
**Lánc endogén ábrázolás**
Pascal megvalósítás:




7. ábra. Lánc endogén ábrázolása.

```
Type
  Elemtip = ???;
  Pozicio = ^Cella;
  Cella = Record
            elem : Elemtip;
            csat : Pozicio
          End;
  Lanc = Pozicio;
```

**Exogén ábrázolás**
Megvalósítás explicit pointer típussal (Pascal).




8. ábra. Lánc exogén ábrázolása.

```
Type
  Elemtip = ???;
  AdatP   = ^Elemtip;
  Pozicio = ^Cella;
  Cella = Record
            adatra : AdatP;
            csat : Pozicio
          End;
  Lanc = Pozicio;
```

Objektum orientált megvalósítás (java).

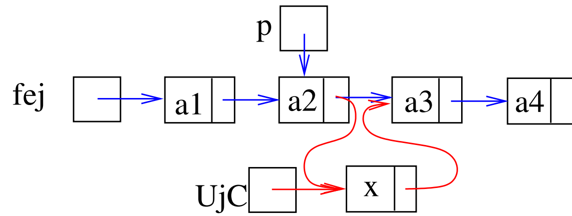
```
public class Lanc<E>{
   public E elem;
   public Lanc<E> csat;

   Lanc(){};
   Lanc(E x, Lanc<E> poz){
      this.csat=poz;
      this.elem=x;
   }
}
```

**Lánc alapműveletek:**
Bővítés a $p$ pozíció után $x$ adattal:

```
p.csat = new Lanc<E>(x, p.csat);
```

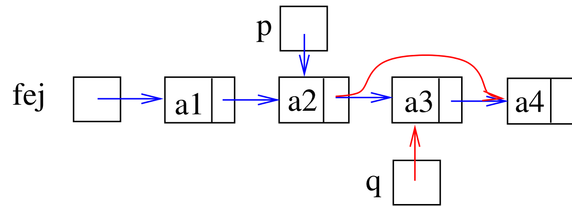
Vagy




9. ábra. Lánc bővítése.

```
Lanc<E> UjC = new Lanc<E>();
UjC.elem = x;
UjC.csat = p.csat;
p.csat = UjC;
```

A $p$-t követő pozíciójú cella törlése:



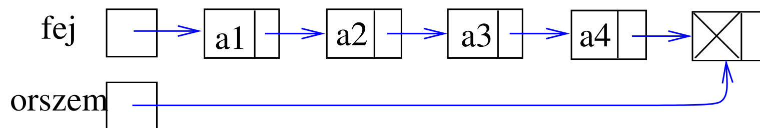

10. ábra. Cella törlése láncból.

```
p.csat = p.csat.csat;
```

Adott $x$ adatot tartalmazó cella keresése:

```
p = fej;                           \\p az első cellára mutasson
while (p != null) && !x.equals(p.elem)
  p = p.csat;                      \\továbblépés
```

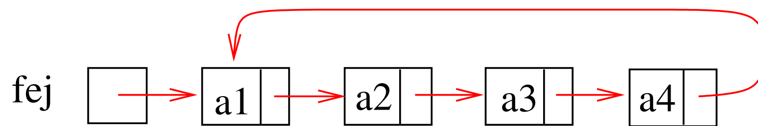
**Lánc ábrázolása őrszemmel.**




11. ábra. Lánc ábrázolása őrszemmel.

```
public class LancO<E>{
   public Lanc<E> fej;
   public Lanc<E> orszem;

   LancO(){
      fej = new Lanc<E>();
      orszem = fej;
   }
}
```



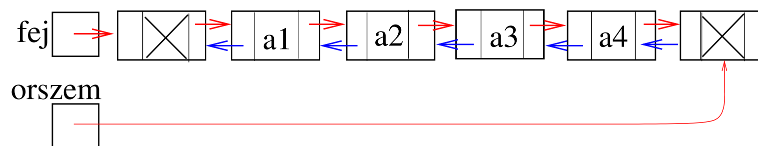

12. ábra. Körlánc ábrázolása.

### 5.2. Körlánc

```
public class Lanc<E>{
   public E elem;
   public Lanc<E> csat;

   Lanc(){};
   Lanc(E x, Lanc<E> poz){
      this.csat=poz;
      this.elem=x;
   }
}
```

### 5.3. Kétirányú lánc



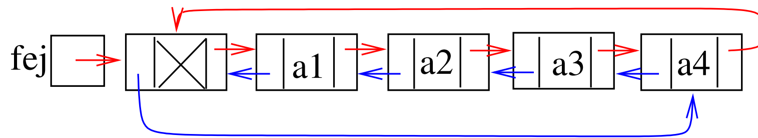

13. ábra. Kétirányú lánc ábrázolása őrszemmel.

```
public class Lanc2<E>{
   public class Cella2<E>{
      public E elem;
      public Cella2<E> elore, hatra;
   }
   public Cella2<E> fej, orszem;

   Lanc2(){
      this.fej = new Cella2<E>();
      this.orszem = new Cella2<E>();
      this.fej.elore = this.orszem;
      this.orszem.hatra = this.fej;
   }
}
```

### 5.4. Kétirányú körlánc

```
public class Lanc2kor<E>{
   public class Cella2<E>{
      public E elem;
      public Cella2<E> elore, hatra;
   }
   public Cella2<E> fej;
```




14. ábra. Kétirányú körlánc ábrázolása.

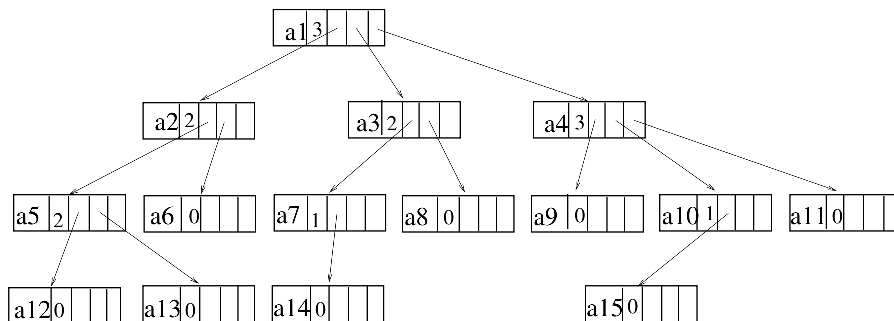
```
   Lanc2kor(){
      fej = new Cella2<E>();
      fej.elore = fej;
      fej.hatra = fej;
   }
}
```

## 5.5. Tömb

Minden (nem assembly szintű) programozási nyelvben közvetlenül megvalósítható.

## 5.6. Fák

### 5.6.1. Kapcsolati tömb ábrázolás




15. ábra. Fa ábrázolása kapcsolati tömbbel.

```
public class FaPontT<E>{
   public E elem;
   public FaPontT<E>[] fiuk;
}
```

Az ábrázolás előnye:
Minden pont i-edik fia közvetlenül (konstans időben) elérhető.
Az ábrázolás hátránya:
Statikus, azaz nem lehet konstans időben bővíteni és törölni.

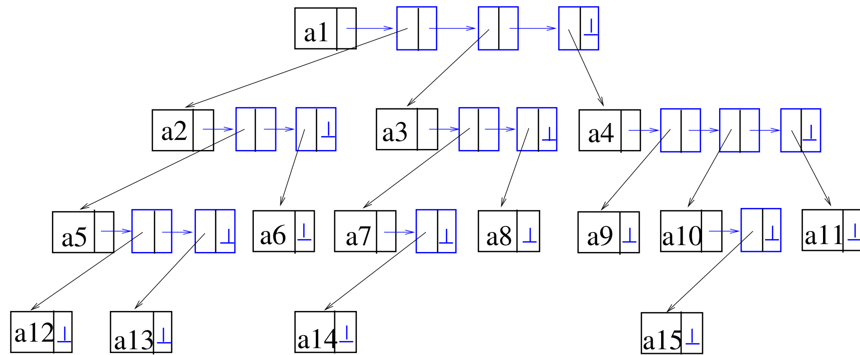
## 5.7. Bináris fák

Az $F$ fát bináris fának nevezzük, ha $(\forall p \in F)(Fok(p) \leq 2)$. Bináris fák ábrázolására a továbbiakban a következő megoldást használjuk.

```
public class BinFaPont<E>{
   public E elem;
   public BinFaPont<E> bal;
   public BinFaPont<E> jobb;
```

```
 //  public BinFaPont<E> apa;
}
```

### 5.7.1. Kapcsolati lánc ábrázolás

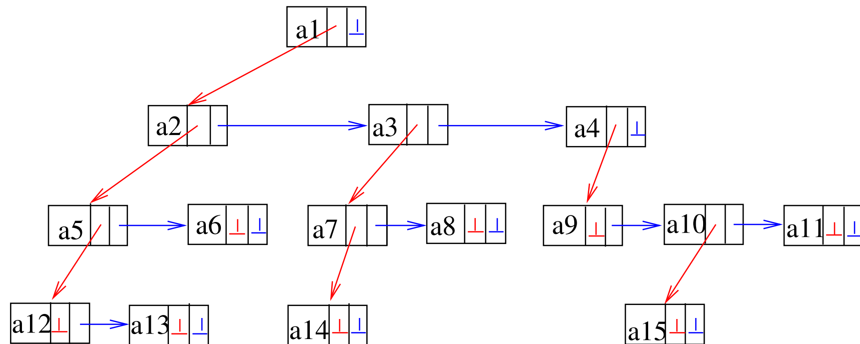



16. ábra. Fa ábrázolása kapcsolati lánccal.

```
public class FaPontL<E>{
   public E elem;
   public Lanc<FaPontL<E>> fiuk;
}
```

Memóriaigény $n$ pontú fára: $n(|Elemtip|+4)+(n-1)*8$

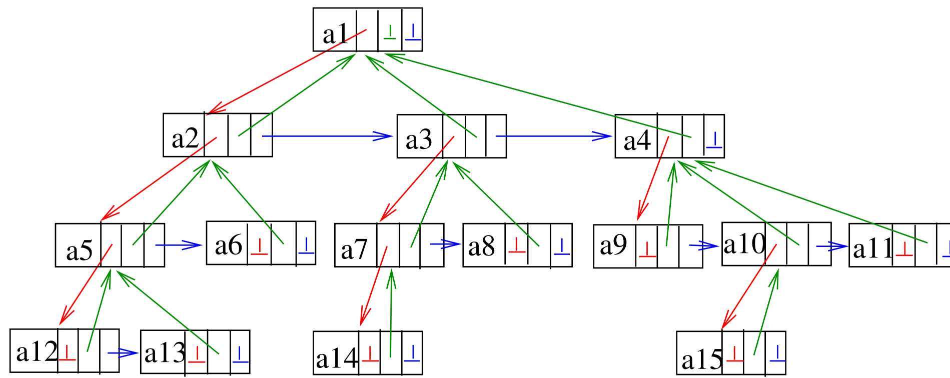
### 5.7.2. Elsőfiú-testvér ábrázolás




17. ábra. Fa ábrázolása elsőfiú-testvér kapcsolattal.

```
public class FaPont<E>{
   public E elem;
   public FaPont<E> elsofiu;
   public FaPont<E> testver;
}
```

Memóriaigény $n$ pontú fára: $n(|Elemtip|+8)$



18. ábra. Fa ábrázolása elsőfiú-testvér-apa kapcsolattal.

#### 5.7.3. Elsőfiú-testvér-apa ábrázolás

```
public class FaPont<E>{
   public E elem;
   public FaPont<E> elsofiu;
   public FaPont<E> testver;
   public FaPont<E> apa;
}
```