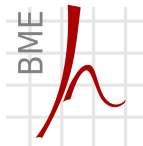


Listák. Bináris fájlok

A programozás alapjai I.



Hálózati Rendszerek és Szolgáltatások Tanszék
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2021. november 9.

Tartalom

- 1 Dinamikus adatszerkezetek
 - Önhivatkozó adatszerkezet
- 2 Egy irányban láncolt listák
 - Definíció
 - Bejárás
 - Verem
 - Beszúrás
 - Törlés
- 3 Fájlkezelés
 - Bináris fájlok
- 4 Két irányban láncolt és strázsás listák
 - Bejárás
 - Beszúrás
 - Törlés
 - Példa
- 5 Speciális listák
 - Fésűs lista

1. fejezet

Dinamikus adatszerkezetek

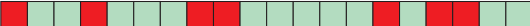
Dinamikus adatszerkezet – motiváció

- Sakkprogramot írunk, melyben a lépések tetszőleges mélységig visszavonhatóak (undo)
- Az undo-lista a játék naplója, elemei a lépések
 - Melyik figura
 - Honnan
 - Hova
 - Kit ütött
- Csak annyi memóriát használhatunk, amennyi feltétlenül szükséges a naplózáshoz.
- A lista maximális hossza csak a játék végére derül ki
- Folytonosan növelnünk (visszavonáskor csökkentenünk) kell a lefoglalt terület méretét.

Dinamikus adatszerkezet – motiváció

- Tömb átméretezése realloc-kal rengeteg fölösleges másolgatást eredményezhet

memória:  – szabad,  – foglalt,  – tömbünk

n=0: 


n=1: 

n=2: 

n=3: 

n=4: 

- Olyan adatszerkezetre van szükségünk, amely nem egybefüggő memóriaterületen tárol, szerkezete dinamikusan változik a program futása közben

n=5: 

Dinamikus adatszerkezet

Dinamikus adatszerkezet:

- mérete és/vagy szerkezete a program futása közben változik
- megvalósítása önhivatkozó adatszerkezettel

Önhivatkozó adatszerkezet

Olyan összetett adatszerkezet, mely önmagára mutató pointereket is tartalmaz

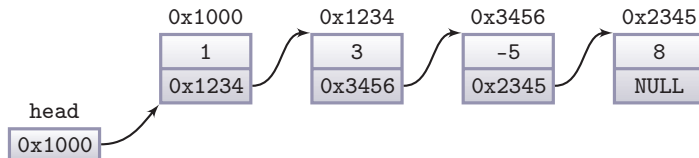
```
1 typedef struct listelem {  
2     int data;                /* a tárolt adat */  
3     struct listelem *next; /* köv. elem címe */  
4 } listelem;
```

- next ugyanolyan struktúrára mutat, mint amelynek ő is része
- a `struct listelem` struktúrát átkereszteltük `listelem`-nek, de `next` deklarációjánál még a hosszú nevet kell használnunk (mert a fordító még nem tudja, minek fogjuk elkeresztelni).

2. fejezet

Egy irányban láncolt listák

Láncolt lista



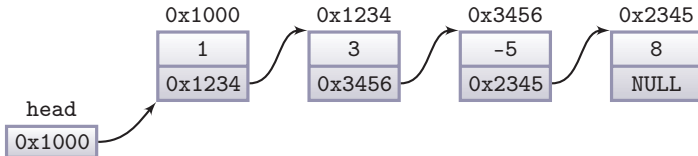
- Azonos listelem típusú változók listája
- Az elemeknek egyenként, dinamikusan foglalunk memóriát
- Az elemek a memóriában nem összefüggő területen helyezkednek el
- Minden elem tárolja a következő elem címét
- Az első elemet a head mutató jelöli ki
- Az utolsó elem nem mutat sehova (NULL)

Láncolt lista

- Az üres lista



- A lista önhivatkozó (rekurzív) adatszerkezet. Minden elem egy listára mutat



Lista vagy tömb

■ A tömb

- annyi memóriát foglal, amennyi az adatok tárolásához szükséges
- egybefüggő memóiahelyet igényel
- akármelyik eleme azonnal elérhető (indexelés)
- adat beszúrása sok másolással jár

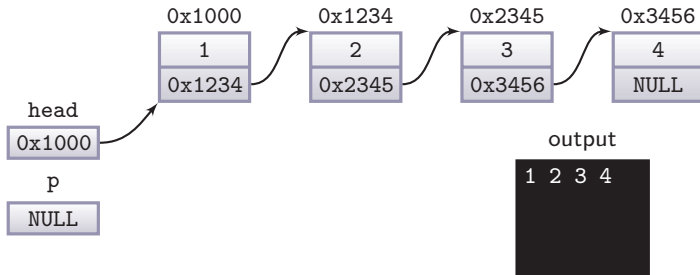
■ A lista

- minden elem tárolja a következő címét, ez sok memóriát foglalhat
- kihasználhatja a töredezett memória lyukait
- csak a következő elem érhető el azonnal
- új adat beszúrása minimális költségű

Lista bejárása

- A bejáráshoz egy segédmutató (p) kell, mely végigfut a listán

```
1 listelem *p = head;  
2 while (p != NULL)  
3 {  
4     printf("%d ", p->data); /* p->data : (*p).data */  
5     p = p->next;           /* nyíl operátor */  
6 }
```



Lista átadása függvénynek

- Mivel a listát a kezdőcím meghatározza, elég azt átadnunk a függvénynek

```
1 void traverse(listelem *head) {  
2     listelem *p = head;  
3     while (p != NULL)  
4     {  
5         printf("%d ", p->data);  
6         p = p->next;  
7     }  
8 }
```

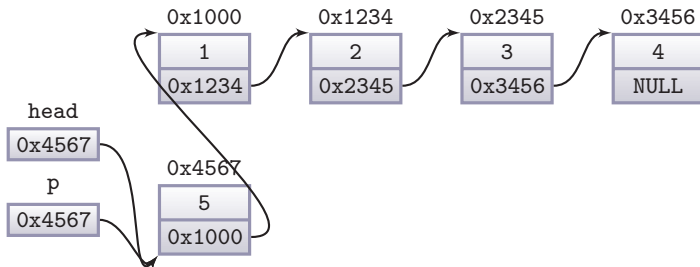
[link](#)

- ugyanaz `for` ciklussal

```
1 void traverse(listelem *head) {  
2     listelem *p;  
3     for (p = head; p != NULL; p = p->next)  
4         printf("%d ", p->data);  
5 }
```

Elem beszúrása lista elejére

```
1 p = (listelem*)malloc(sizeof(listelem));  
2 p->data = 5;  
3 p->next = head;  
4 head = p;
```



Elem beszúrása lista elejére függvénnel

- Mivel beszúráskor a kezdőcím változik, azt vissza kell adnunk

```
1 listelem *push_front(listelem *head, int d)
2 {
3     listelem *p = (listelem*)malloc(sizeof(listelem));
4     p->data = d;
5     p->next = head;
6     head = p;
7     return head;
8 }
```

[link](#)

- A függvény használata

```
1 listelem *head = NULL;          /* üres lista */
2 head = push_front(head, 2);      /* head változik! */
3 head = push_front(head, 4);
```

Elem beszúrása lista elejére függvényrel

- Másik lehetőségként a kezdőcímet cím szerint adjuk át

```
1 void push_front(listelem **head, int d)
2 {
3     listelem *p = (listelem*)malloc(sizeof(listelem));
4     p->data = d;
5     p->next = *head;
6     *head = p; /* *head változik, ez nem vész el */
7 }
```

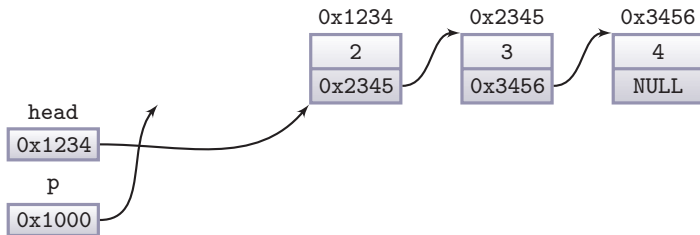
[link](#)

- Ekkor a függvény használata

```
1 listelem *head = NULL; /* üres lista */
2 push_front(&head, 2); /* címmel hívás */
3 push_front(&head, 4);
```

Elem törlése lista elejéről

```
1 p = head;  
2 head = head->next;  
3 free(p);
```



Elem törlése lista elejéről függvénnel

```
1 listelem *pop_front(listelem *head)
2 {
3     if (head != NULL) /* nem üres */
4     {
5         listelem *p = head;
6         head = head->next;
7         free(p);
8     }
9     return head;
10 }
```

[link](#)

- Az üres listára külön figyelünk kell
- Természetesen itt is használhatnánk a head címmel hívott változatot

Verem

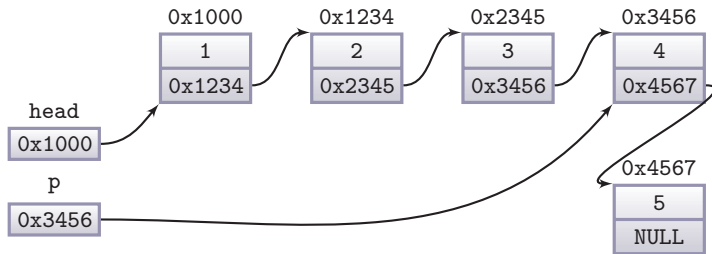
- Ami eddig elkészült, már elég az undo-lista tárolásához

```
1 listelem *head = NULL;           /* üres lista */
2 head = push_front(head, 2);      /* lépés */
3 head = push_front(head, 4);      /* lépés */
4 printf("Az utoljára betett elem: %d\n", head->data);
5 head = pop_front(head);          /* undo */
6 head = push_front(head, 5);      /* lépés */
7 head = pop_front(head);          /* undo */
8 head = pop_front(head);          /* undo */
```

- A verem (LIFO: Last In, First Out)
- A legutoljára berakott elemhez férünk hozzá először

Elem beszúrása lista végére

```
1 for (p = head; p->next != NULL; p = p->next);  
2 p->next = (listelem*)malloc(sizeof(listelem));  
3 p->next->data = 5;  
4 p->next->next = NULL;
```



- Üres listára a `p->next != NULL` vizsgálat értelmetlen, azt külön kell kezelnünk!

Elem beszúrása rendezett listába

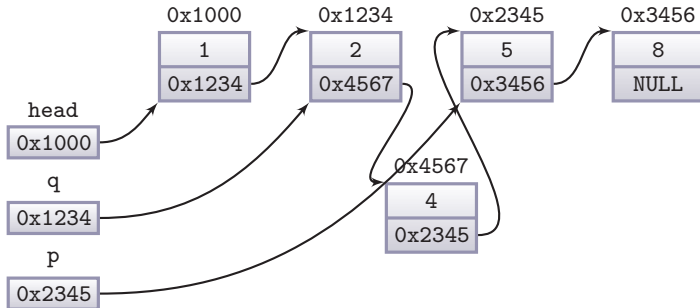
- Sokszori bejárás és feldolgozás esetén érdemes rendeznünk az adatokat
- Tömbök:
 - egyetlen elem áthelyezése rengeteg adatmozgatással jár
 - feltöltjük a tömböt, majd utólag rendezünk
- Listák:
 - egyetlen elem áthelyezése csak láncolgatással jár, az elemek a memóriában ugyanott maradnak
 - érdemes eleve rendezve építenünk
- Az új elemet az első nála nagyobb elem elé kell beszúrunk
- A jelenlegi szerkezetben minden elem csak „maga mögé lát”, nem tudunk elem elé szűrni
- Két mutatóval járjuk be a listát, az egyik mindig eggyel lemarad
- A lemaradó mutató mögé szúrunk

Elem (4) beszúrása rendezett listába

```

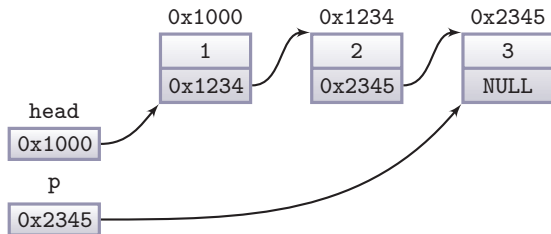
1  q = head;  p = q->next;
2  while (p != NULL && p->data <= data) { /* rövidzár */
3      q = p;  p = p->next;
4  }
5  q->next = (listelem*)malloc(sizeof(listelem));
6  q->next->data = 4;
7  q->next->next = p;

```



Elem törlése lista végéről

```
1 p = head;  
2 while (p->next->next != NULL)  
3     p = p->next;  
4 free(p->next);  
5 p->next = NULL;
```

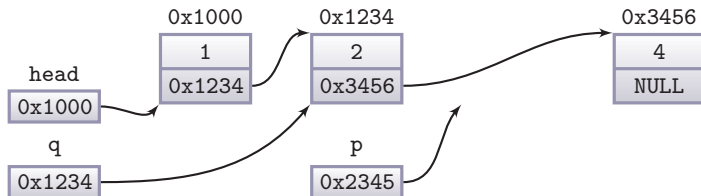


- Ha a lista üres, vagy egy eleme van, a `p->next->next` kifejezés értelmetlen

Adott elem törlése listából

■ A data = 3 elem törlése

```
1 q = head; p = head->next;
2 while (p != NULL && p->data != data) {
3     q = p; p = p->next;
4 }
5 if (p != NULL) { /* megvan */
6     q->next = p->next;
7     free(p);
8 }
```



■ Ha a lista üres, vagy az első elemet kell törölnünk, nem működik

Teljes lista törlése

```
1 void dispose_list(listelem *head)
2 {
3     while (head != NULL)
4         head = pop_front(head);
5 }
```

[link](#)

3. fejezet

Fájlkezelés

Bináris fájlok

- Bináris fájl: A memória tartalmának bithű másolata egy fizikai hordozón
- A tárolt adat természetesen belsőábrázolás-függő
- Csak akkor használjuk, ha a szöveges tárolás nagyon ésszerűtlen lenne – már a nagy háziban sem kötelező elem 😊
- Fájlnyitás és fájlzárás a szöveges fájlokhoz hasonlóan, csak a mode sztringben szerepelnie kell a **b** karakternek¹

mode		leírás
"rb"	read	olvasásra, a fájlnek léteznie kell
"wb"	write	írásra, felülír, ha kell, újat hoz létre,
"ab"	append	írásra, végére ír, ha kell, újat hoz létre

¹Az analógia kedvéért szöveges fájlknál bevett szokás a **t** (text) szerepeltetése, de ezt az fopen figyelmen kívül hagyja

Bináris fájl írása olvasása

```
size_t fwrite (void *ptr, size_t size,  
               size_t count, FILE *fp);
```

- A ptr címtől count számú, egyenként size méretű, folytonosan elhelyezkedő elemet ír az fp azonosítójú fájlba
- Visszatérési érték a beírt **elemek** száma

```
size_t fread (void *ptr, size_t size,  
              size_t count, FILE *fp);
```

- A ptr címre count számú, egyenként size méretű elemet olvas az fp azonosítójú fájlból
- Visszatérési érték a kiolvasott **elemek** száma

Bináris fájlok – példa

- Az alábbi dog_array tömb 5 kutyát tárol

```
1 typedef enum { BLACK, WHITE, RED } color_t;
2
3 typedef struct {
4     char name[11];      /* név max 10 karakter + lezárás */
5     color_t color;      /* szín */
6     int nLegs;          /* lábak száma */
7     double height;      /* magasság */
8 } dog;
9
10 dog dog_array[] = /* 5 kutya tömbje */
11 {
12     { "blöki", RED, 4, 1.12 },
13     { "cézár", BLACK, 3, 1.24 },
14     { "buksi", WHITE, 4, 0.23 },
15     { "spider", WHITE, 8, 0.45 },
16     { "mici", BLACK, 4, 0.456 }
17 };
```

[link](#)

Bináris fájlok – példa

- A dog_array tömb kiírása bináris fájlba enyire egyszerű!

```
1 fp = fopen("dogs.dat", "wb"); /* hibakezelés!!! */
2 if (fwrite(dog_array, sizeof(dog), 5, fp) != 5)
3 {
4     /* hibajelzés */
5 }
6 fclose(fp); /* ide is!!! */
```

- A dog_array tömb visszaolvasása sem bonyolultabb

```
1 dog dogs[5]; /* tárhely foglалás */
2 fp = fopen("dogs.dat", "rb");
3 if (fread(dogs, sizeof(dog), 5, fp) != 5)
4 {
5     /* hibajelzés */
6 }
7 fclose(fp);
```

Bináris fájlok – példa

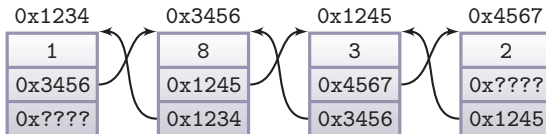
- Álljunk ellen a csábításnak!
- Ha egy másik gépen a dog struktúra bármely tagjának ábrázolása eltérő, a kimentett adatokat ott nem tudjuk visszaolvasni
- Az átgondolatlanul kimentett bináris fájlok a programot hordozhatatlanná teszik
- Az átgondolt kimentés természetesen jóval bonyolultabb
 - 1 Megállapodunk az ábrázolásban
 - melyik bájt az LSB?
 - kettes komplement?
 - hány bites a mantissa?
 - struktúra elemei szóhatárra illesztettek? És az mekkora?
 - ...
 - 2 Az adatokat konvertáljuk, majd kiírjuk

4. fejezet

Két irányban láncolt és strázsás listák

Kétirányú láncolás

- A két irányban láncolt lista minden eleme a következő és az előző elemre is hivatkozik



- C nyelvi megvalósítás

```

1 typedef struct listelem {
2     int data;
3     struct listelem *next;
4     struct listelem *prev;
5 } listelem;

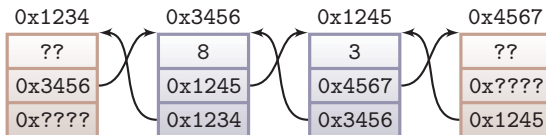
```

[link](#)

- A kétirányú összefűzés lehetővé teszi, hogy nemcsak elem mögé, hanem elem elé is beszúrhatunk

Strázsák

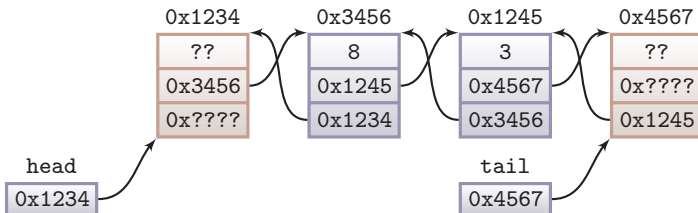
- A strázsás listát egyik vagy mindkét végén érvénytelen elem, a strázsa (őrszem, sentinel) zárja



- A strázsa ugyanolyan típusú, mint a közbülső listaelemek
- A strázsában tárolt adat nem része a listának
 - értéke sokszor (rendezetlen listában) érdektelen
 - rendezett listában a strázsa adata lehet a garantáltan legnagyobb vagy legkisebb elem
- A kétstrázsás listánk haszna:
 - a beszúrás – még üres listában is – mindig két elem közé történik
 - a törlés mindig két elem közül történik
 - nem kell külön figyelniük a kivételekre

Két irányban láncolt, kétstrázsás lista

- A strázsákra a head és tail mutatók mutatnak



- ezeket célszerűen egységbe zárjuk, ez az egység testesíti meg a listát

```
1 typedef struct {
2     listelem *head, *tail;
3 } list;
```

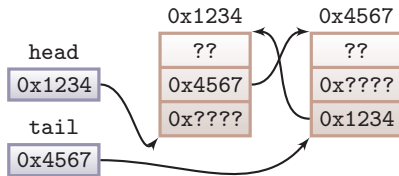
[link](#)

- A strázsákat csak a lista megszüntetésekor töröljük, list tagjai használat közben állandóak

Üres lista létrehozása

- A `create_list` függvény üres listát hoz létre

```
1 list create_list(void)
2 {
3     list l;
4     l.head = (listelem*)malloc(sizeof(listelem));
5     l.tail = (listelem*)malloc(sizeof(listelem));
6     l.head->next = l.tail;
7     l.tail->prev = l.head;
8     return l;
9 }
```

[link](#)

Lista bejárása

- Az isempty függvény ellenőrzi, hogy a lista üres-e

```
1 int isempty(list l)
2 {
3     return (l.head->next == l.tail);
4 }
```

[link](#)

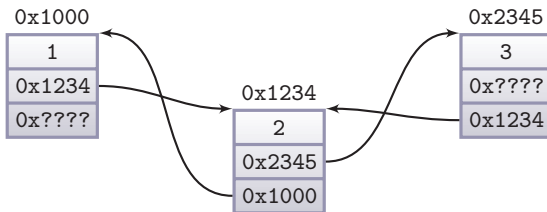
- Lista bejárása: a p mutatóval head->next-től tail-ig megyünk.

```
1 void print_list(list l)
2 {
3     listelem *p;
4     for (p = l.head->next; p != l.tail; p = p->next)
5         printf("%3d", p->data);
6 }
```

[link](#)

Elem becsatolása két szomszédos listaelem közé

```
1 void insert_between(listelem *prev, listelem *next,  
2     int d)  
3 {  
4     listelem *p = (listelem*)malloc(sizeof(listelem));  
5     p->data = d;  
6     p->prev = prev;  
7     prev->next = p;  
8     p->next = next;  
9     next->prev = p;  
10 }
```

[link](#)

Elem beszúrása listába

■ lista elejére

```
1 void push_front(list l, int d) {  
2     insert_between(l.head, l.head->next, d);  
3 }
```

[link](#)

■ lista végére (nem figyeljük, hogy üres-e)

```
1 void push_back(list l, int d) {  
2     insert_between(l.tail->prev, l.tail, d);  
3 }
```

[link](#)

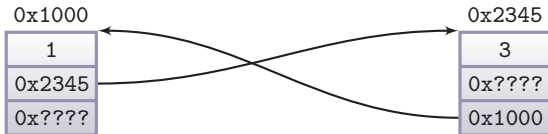
■ rendezett listába (nem kell lemaradó mutató)

```
1 void insert_sorted(list l, int d) {  
2     listelem *p = l.head->next;  
3     while (p != l.tail && p->data <= d)  
4         p = p->next;  
5     insert_between(p->prev, p, d);  
6 }
```

[link](#)

Elem törlése nem üres listából

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```

[link](#)

Elem törlése a listából

■ lista elejéről (a törölt adatot visszaadjuk)

```
1 int pop_front(list l)
2 {
3     int d = l.head->next->data;
4     if (!isempty(l))
5         delete(l.head->next);
6     return d; /* üres esetén strázsaszemét */
7 }
```

[link](#)

■ lista végéről

```
1 int pop_back(list l)
2 {
3     int d = l.tail->prev->data;
4     if (!isempty(l))
5         delete(l.tail->prev);
6     return d; /* üres esetén strázsaszemét */
7 }
```

[link](#)

Elem törlése a listából

■ adott elem törlése

```
1 void remove_elem(list l, int d)
2 {
3     listelem *p = l.head->next;
4     while (p != l.tail && p->data != d)
5         p = p->next;
6     if (p != l.tail)
7         delete(p);
8 }
```

[link](#)

■ teljes lista törlése strázsákkal együtt

```
1 void dispose_list(list l) {
2     while (!isempty(l))
3         pop_front(l);
4     free(l.head);
5     free(l.tail);
6 }
```

[link](#)

Használat

■ Egy egyszerű alkalmazás

```
1 list l = create_list();  
2 push_front(l, -1);  
3 push_back(l, 1);  
4 insert_sorted(l, -3);  
5 insert_sorted(l, 8);  
6 remove_elem(l, 1);  
7 print_list(l);  
8 dispose_list(l);
```

[link](#)

A tárolt adat

- A listákban természetesen nem csak `int`-eket tárolhatunk
- Érdemes szétválasztani a tárolt adatot és a lista mutatóit az alábbiak szerint

```
1 typedef struct {  
2     char name[30];  
3     int age;  
4     ...  
5     double height;  
6 } data_t;  
7  
8 typedef struct listelem {  
9     data_t data;  
10    struct listelem *next, *prev;  
11 } listelem;
```

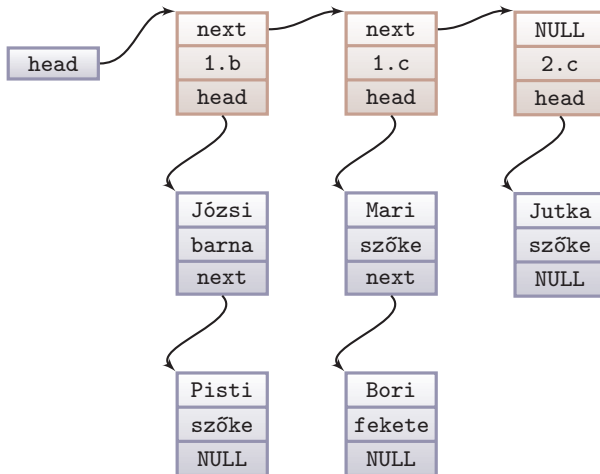
- Ha a tárolt adat önálló struktúra típus, akkor az `int`-hez hasonlóan egyetlen utasítással értékül adhatjuk, szerepelhet függvényparaméterként, visszatérési típusként

5. fejezet

Speciális listák

Fésűs lista

- Osztályok listája, minden osztály a tanulók listáját tartalmazza.



Fésűs lista – az adatok célszerű szétválasztásával

```
1 typedef struct {
2     char name[50];           /* neve */
3     color_t hair_color;      /* hajszíne (typedef) */
4 } student_t;                /* hallgató adat */
5
6 typedef struct student_elem {
7     student_t student;       /* a hallgató maga */
8     struct student_elem *next; /* láncolás */
9 } student_elem;             /* hallgató listaelem */
10
11 typedef struct {
12     char name[10];           /* osztály neve */
13     student_elem *head;      /* hallgatók listája */
14 } class_t;                   /* osztály adat */
15
16 typedef struct class_elem {
17     class_t class;           /* az osztály maga */
18     struct class_elem *next; /* láncolás */
19 } class_elem;                /* osztály listaelem */
```

Köszönöm a figyelmet.