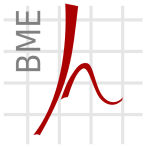
Bináris fájlok. Haladó listakezelés

A programozás alapjai I.



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

2020. november 9.

Tartalom

1. 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` karakternek[1]

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

[1] Az analógia kedvéért szöveges fájloknál 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 foglalás */
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 bit az LSB?
     - kettes komplemens?
     - hány bites a mantissza?
     - struktúra elemei szóhatárra illesztettek? És az mekkora?
     - stb
  2. Az adatokat konvertáljuk, majd kiírjuk

Bináris vs szöveges

- Csináljuk inkább szövegesen, mindenki jobban jár!
- A dog_array tömb kiírása szöveges fájlba

```
1 for (i = 0; i < 5; ++i) {
2   dog d = dog_array[i];
3   fprintf(fp, "%s,%u,%d,%f\n",
4     d.name, d.color, d.nLegs, d.height);
5 }
```

- A dog_array tömb beolvasása szöveges fájlból[2]

```
1 dog dogs[5]; /* tárhely foglalás */
2 for (i = 0; i < 5; ++i) {
3   dog d;
4   fscanf(fp, "%s,%u,%d,%lf",
5     d.name, &d.color, &d.nLegs, &d.height);
6   dogs[i] = d;
7 }
```

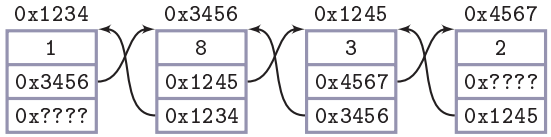
[2] feltételezzük, hogy a kutya neve nem tartalmaz whitespace karaktert

2. fejezet

Két irányban láncolt és strázsás listák

Kétirányú láncolás

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



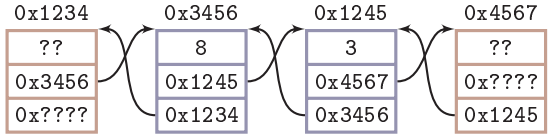
- C nyelvi megvalósítás

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

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

Strázsák

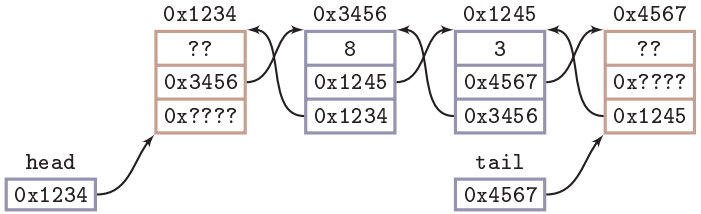
- A strázsás listát egyik vagy mindkét végén érvénytelen elem, a strázsa (őrszem, sentinel) zárja



- 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 figyelnünk 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

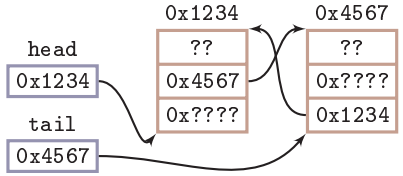
```
typedef struct {
  listelem *head, *tail;
} list;
```

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

Üres lista létrehozása

- A create_list függvény üres listát hoz létre

```
list create_list(void)
{
  list l;
  l.head = (listelem*)malloc(sizeof(listelem));
  l.tail = (listelem*)malloc(sizeof(listelem));
  l.head->next = l.tail;
  l.tail->prev = l.head;
  return l;
}
```



Lista bejárása

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

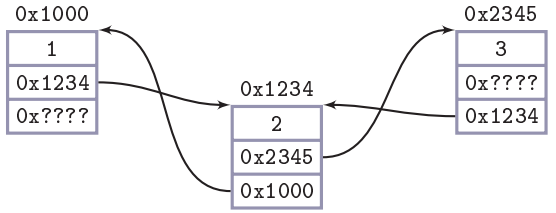
```
int isempty(list l)
{
  return (l.head->next == l.tail);
}
```

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

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


Elem becsatolása két szomszédos listaelem közé

```
void insert_between(listelem *prev, listelem *next,
  int d)
{
  listelem *p = (listelem*)malloc(sizeof(listelem));
  p->data = d;
  p->prev = prev;
  prev->next = p;
  p->next = next;
  next->prev = p;
}
```



Elem beszúrása listába

- lista elejére

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

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

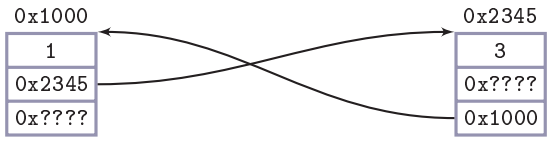
```
1 void push_back(list l, int d) {
2   insert_between(l.tail->prev, l.tail, d);
3 }
```

- 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 }
```


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 }
```



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 }
```

- 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 }
```


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 }
```

- 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 }
```

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);
```


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

3. fejezet

Speciális listák

FIFO

FIFO-tároló

FIFO (First In First Out) – az elemekhez a beszúrás sorrendjében férünk hozzá

- Tipikus alkalmazás: várakozási sorok, az elemeket érkezési sorrendben dolgozzuk fel
- Megvalósítás: pl. az előző listával
  - beszúrásra csak a `push_front`
  - kivételre csak a `pop_back`

  függvényeket használjuk

Verem

Verem (Stack/LIFO-tároló)

LIFO (Last In First Out) – az elemekhez a beszúrással ellentétes sorrendben férünk hozzá

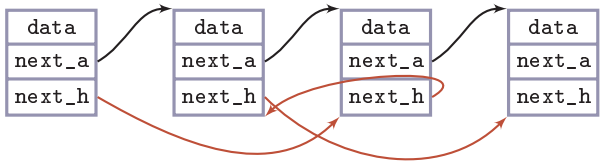
- Tipikus alkalmazás: „undo”-lista tárolása, függvényhívás visszatérési címeinek tárolása
- Megvalósítás: pl. az előző listával
  - beszúrásra csak a `push_front`
  - kivételre csak a `pop_front`

  függvényeket használjuk

Több szempont szerint rendezett lista

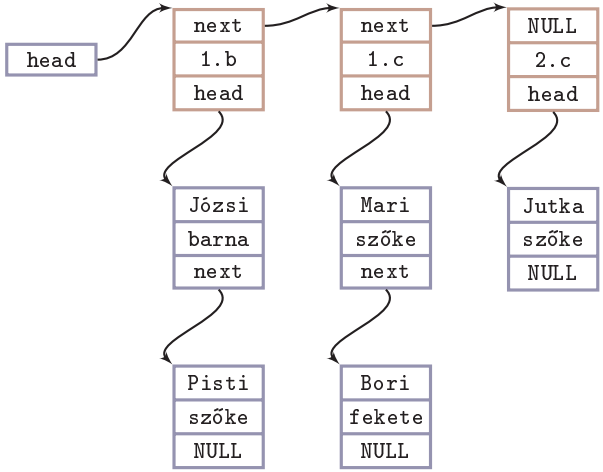
- Egyszerre több szempont szerint rendezett lista elemeinek típusa

```
1 typedef struct person {
2   data_t data;                /* személy adatai */
3   struct person *next_age;    /* köv. legfiatalabb */
4   struct person *next_height; /* köv. legalacsonyabb */
5 } person;
```



Fésűs lista

- Osztályok listája, minden osztály a tanulók listáját tartalmazza.



Fésűs lista – deklarációk

```
1  typedef struct student_elem {
2    char name[50];              /* neve */
3    color_t hair_color;         /* hajszíne (typedef) */
4    struct student_elem *next; /* láncolás */
5  } student_elem;       /* hallgató listaelem */
6
7  typedef struct class_elem {
8    char name[10];              /* osztály neve */
9    student_elem *head;         /* hallgatók listája */
10   struct class_elem *next;    /* láncolás */
11 } class_elem;         /* osztály listaelem */
```

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 */
```


4. fejezet

Többdimenziós tömbök

Többdimenziós tömbök

1D tömb Azonos típusú elemek a memóriában egymás mellett tárolva

2D tömb Azonos méretű és típusú 1D tömbök a memóriában egymás mellett tárolva

3D tömb Azonos méretű és típusú 2D tömbök a memóriában egymás mellett tárolva

... ...

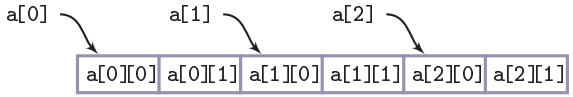
Kétdimenziós tömbök

- 2D tömb deklarációja:

```
1 char a[3][2]; /* 3 soros két oszlopos karaktertömb */
2               /* 2 elemű 1D tömbök 3 elemű tömbje */
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

- C-ben sorfolytonos tárolás, vagyis a hátsó index fut gyorsabban



- a[0], a[1] és a[2] 2 elemű 1D tömbök

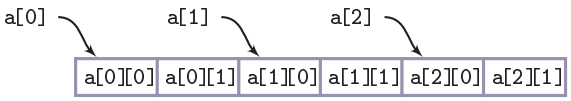
Kétdimenziós tömb átvétele soronként

- 1D tömb (sor) feltöltése adott elemmel

```
1  void fill_row(char row[], size_t size, char c)
2  {
3    size_t i;
4    for (i = 0; i < size; ++i)
5      row[i] = c;
6  }
```

- 2D tömb feltöltése soronként

```
1  char a[3][2];
2  fill_row(a[0], 2, 'a'); /* 0. sor csupa 'a' */
3  fill_row(a[1], 2, 'b'); /* 1. sor csupa 'b' */
4  fill_row(a[2], 2, 'c'); /* 2. sor csupa 'c' */
```



Kétdimenziós tömb átvétele egyben

- átvétel 2D tömbként – csak ha az oszlopok száma ismert

```
1  void print_array(char array[][2], size_t nrows)
2  {
3    size_t row, col;
4    for (row = 0; row < nrows; ++row)
5    {
6      for (col = 0; col < 2; ++col)
7        printf("%c", array[row][col]);
8      printf("\n");
9    }
10 }
```

- A függvény használata

```
1  char a[3][2];
2  ...
3  print_array(a, 3);       /* 3 soros tömb kiírása */
```


Kétdimenziós tömb átvétele egyben

- 2D tömb átvétele mutatóként

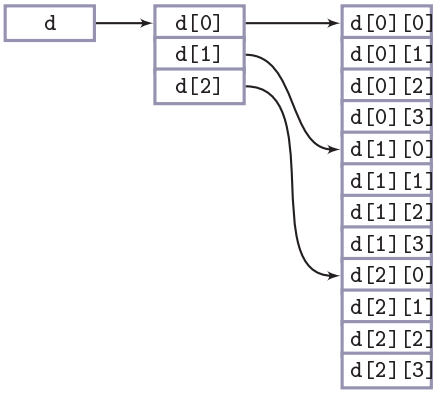
```
1  void print_array(char *array, int nrows, int ncols)
2  {
3    int row, col;
4    for (row = 0; row < nrows; ++row)
5    {
6      for (col = 0; col < ncols; ++col)
7        printf("%c", array[row*ncols+col]);
8      printf("\n");
9    }
10 }
```

- A függvény használata

```
1  char a[3][2];
2  ...
3  print_array((char *)a, 3, 2); /* 3 sor 2 oszlop */
```

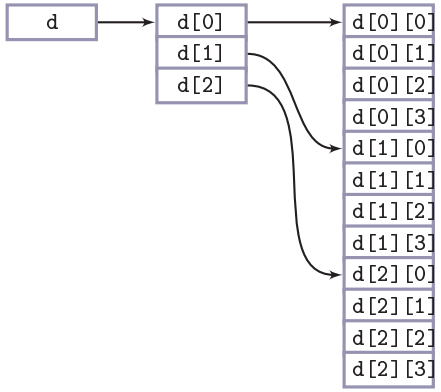

2D dinamikus tömb

Foglaljunk dinamikusan kétdimenziós tömböt, melyet a szokásos módon, d[i][j] indexeléssel használhatunk




```
1  double **d =(double**)malloc(3*sizeof(double*));
2  d[0] = (double*)malloc(3*4*sizeof(double));
3  for (i = 1; i < 3; ++i)
4    d[i] = d[i-1] + 4;
```

2D dinamikus tömb



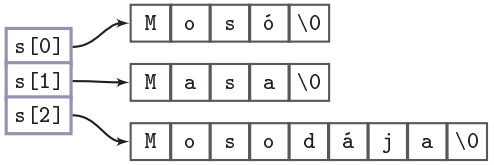
A tömb felszabadítása

```
1 free(d[0]);
2 free(d);
```


Mutatótömb

- Mutatótömb definiálása és átadása függvénynek

```
1 char *s[3] = {"Mosó", "Masa", "Mosodája"};
2 print_strings(s, 3);
```



- Mutatótömb átvétele függvénnyel

```
1 void print_strings(char *strings[], size_t size)
2 /*                  char **strings is lehet        */
3 {
4   size_t i;
5   for (i = 0; i < size; ++i)
6     printf("%s\n", strings[i]);
7 }
```