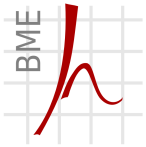
Szöveges fájlok. Láncolt listák

A programozás alapjai I.



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

2020. november 2.

Tartalom

1. fejezet

Fájlkezelés

Fájlok

Fájl

Fizikai hordozón (merevlemez, CD, USB drive) tárolt adat

- A fájlba kimentett adat nem vész el a program futása után, következő futáskor visszaolvasható
- A különböző hordozókon tárolt fájlokat egységes felületen kezeljük
- Fájlkezelés:
  1. Fájl megnyitása
  2. Adatok írása / olvasása
  3. Fájl bezárása
- Kétféle fájltípus:
  - Szöveges fájl
  - Bináris fájl

Szöveges vs. Bináris

Szöveges fájl – szöveget tartalmaz, sorokra tagolódik
- `txt`, `c`, `html`, `xml`, `rtf`, `svg`

Bináris fájl – tetszőleges struktúrájú binárisan kódolt adatot tartalmaz
- `exe`, `wav`, `mp3`, `jpg`, `avi`, `zip`

- Amíg nem túl ésszerűtlen, ragaszkodjunk az emberbarát szöveges tároláshoz.
- Nagy előny, ha adatainkat nemcsak programok, hanem emberek is értik, szerkeszthetik.

Szöveges fájlba írás

```c
#include <stdio.h> /* fopen, fprintf, fclose */
int main(void)
{
  FILE *fp;
  int status;

  fp = fopen("hello.txt", "w");  /* fájlnyitás */
  if (fp == NULL)                /* nem sikerült */
    return 1;

  fprintf(fp, "Szia, világ!\n");  /* beírás */

  status = fclose(fp);            /* lezárás */
  if (status != 0)
    return 1;

  return 0;
}
```


Fájl megnyitása

```c
FILE *fopen(char *fname, char *mode);
```

- Megnyitja az `fname` sztringben megadott nevű fájlt a `mode` sztringben megadott módon
- Szöveges fájlokhoz használt fontosabb módok:

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

- visszatérési érték mutató egy `FILE` struktúrára, ez a fájlpointer
- Ha a fájlnyitás nem sikeres, nullpointerrel tér vissza

Fájl bezárása

```c
int fclose(FILE *fp);
```

- Lezárja az `fp` fájlpointerrel hivatkozott fájlt
- Ha a lezárás sikeres[1], 0 értékkel, egyébként `EOF`-fal tér vissza

[1] fájlzárás lehet sikertelen. Pl. valaki kihúzta a pendrive-ot, miközben írtunk.

stdoutra / szöveges fájlba / sztringbe írás

```
int  printf(           char *control, ...);
int fprintf(FILE *fp,  char *control, ...);
int sprintf(char *str, char *control, ...);
```

- A `control` sztringben meghatározott szöveget írja a
  - képernyőre
  - `fp` azonosítójú (már írásra megnyitott) szöveges fájlba
  - `str` című (elegendően hosszú) sztringbe
- Visszatérési érték a beírt karakterek száma[2], hiba esetén negatív

[2] Ha sztringbe írunk, automatikusan beírja a lezáró 0-t is, de nem számolja bele a kimenetbe

stdinről / szöveges fájlból / sztringből olvasás

```
int  scanf(           char *control, ...);
int fscanf(FILE *fp,  char *control, ...);
int sscanf(char *str, char *control, ...);
```

- A `control` sztringben meghatározott formátum szerint olvas a
  - billentyűzetről
  - `fp` azonosítójú (már olvasásra megnyitott) szöveges fájlból
  - `str` kezdőcímű sztringből
- Visszatérési érték a kiolvasott elemek száma, hiba esetén negatív

Szöveges fájlból olvasás

Írjunk programot, amely szöveges fájl tartalmát kiírja a képernyőre

```
1  #include <stdio.h>
2  int main()
3  {
4    char c;
5    FILE *fp = fopen("fajl.txt", "r"); /* fájlnyitás */
6    if (fp == NULL)
7      return -1; /* sikertelen volt */
8
9    /* olvasás, amíg sikeres (1 karakter jött) */
10   while (fscanf(fp, "%c", &c) == 1)
11     printf("%c", c);
12
13   fclose(fp); /* lezárás */
14   return 0;
15 }
```

- Jól figyeljük meg, hogyan olvasunk fájl végéig!

Szöveges fájlból olvasás

Egy szöveges fájl kétdimenziós pontok koordinátáit tartalmazza, minden sora az alábbi formátumú

`x:1.2334, y:-23.3`

Írjunk programot, mely beolvassa és feldolgozza a koordinátákat!

```
1  FILE *fp;
2  double x, y;
3  ...
4  /* olvasás, amíg sikeres (2 számot olvastunk) */
5  while (fscanf(fp, "x:%lf, y:%lf", &x, &y) == 2)
6  {
7    /* feldolgozás */
8  }
```

- Ismét jól figyeljük meg, hogyan olvasunk fájl végéig!

Billentyűzet? Monitor?

```c
scanf("%c", &c);
printf("%c", c);
```

- A fenti kódrészlet nem közvetlenül a billentyűzetről olvas és monitorra ír, hanem a standard inputról (stdin) olvas, és a standard outputra (stdout) ír
- stdin és stdout szöveges fájlok
- Az operációs rendszeren múlik, hogy milyen periféria vagy egyéb fájl van hozzájuk rendelve
- Alapértelmezés az ábra szerint
  - billentyűzet (konzol programon keresztül) → stdin
  - stdout → (konzol programon keresztül) monitor

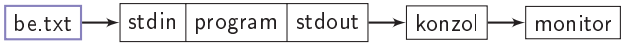
Átirányítás

- Ha a programot az alábbi módon indítjuk, a standard output nem a monitorra megy, hanem a `ki.txt` szöveges fájlba

```
c:\>prog.exe > ki.txt
```

- A standard input is átirányítható szöveges fájlra

```
c:\>prog.exe < be.txt
```



- Természetesen együtt is lehet

```
c:\>prog.exe < be.txt > ki.txt
```


stdin és stdout

- Az stdin és stdout szöveges fájlok automatikusan nyitva vannak program indításakor
- az alábbi kódrészletek ekvivalensek

```c
char c;
printf("Hello");
scanf("%c", &c);
printf("%c", c);
```

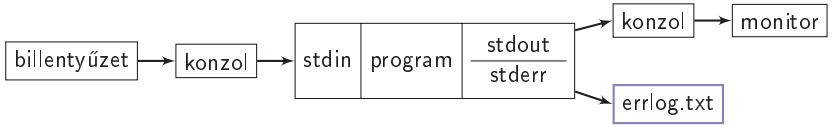
```c
char c;
fprintf(stdout, "Hello");
fscanf(stdin, "%c", &c);
fprintf(stdout, "%c", c);
```

- Ha szöveges fájlból szöveges fájlba dolgozó programot írunk, fájlnyitás helyett használjuk a standard be- és kimenetet és az operációs rendszer átirányítási lehetőségeit
- Konzolról is olvashatunk fájl végéig, amit Ctrl+Z (windows) vagy Ctrl+D (linux) leütésével szimulálhatunk.

stdout és stderr

- A program kimenete és hibaüzenetei is különválaszthatóak a `stderr` szabványos hibakimenet használatával

```
c:\>prog.exe 2> errlog.txt
```




```c
if (error)
{
  /* felhasználónak, ami rá tartozik */
  printf("Kérem, kapcsolja ki\n");
  /* hibakimenetre részletes információ */
  fprintf(stderr, "61. kódú hiba\n");
}
```

2. fejezet

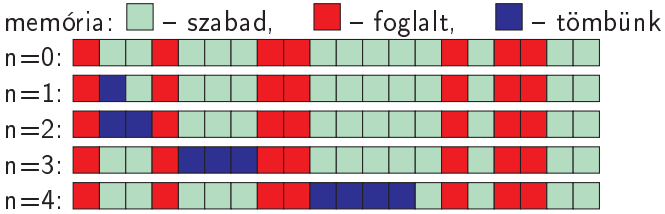
Dinamikus adatszerkezetek

Dinamikus adatszerkezet – motiváció

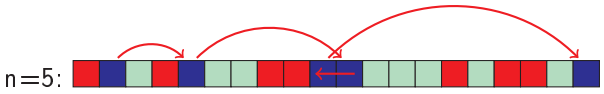
- Sakkprogramot írunk, melyben a lépések tetszőleges mélységig visszavonhatóak (undo)
- Az undo-lista a játék naplója, elemei a lépések
  - Melyik figura
  - Honnan
  - Hova
  - Kit ütött
- Csak annyi memóriát használhatunk, amennyi feltétlenül szükséges a naplózáshoz.
- A lista maximális hossza csak 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



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



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

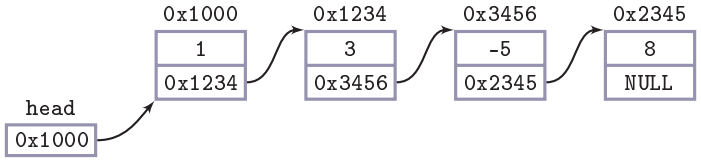
```
1 typedef struct listelem {
2   int data;               /* a tárolt adat */
3   struct listelem *next; /* köv. elem címe */
4 } listelem;
```

- `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).

3. fejezet

Egy irányban láncolt listák

Láncolt lista

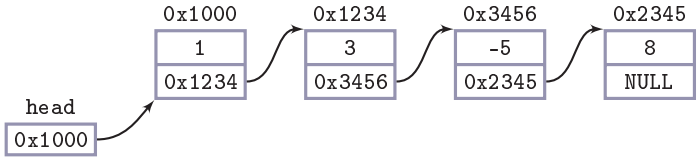


- Azonos `listelem` típusú változók listája
- Az elemeknek egyenként, dinamikusan foglalunk memóriát
- Az elemek a memóriában nem összefüggő területen helyezkednek el
- Minden elem tárolja a következő elem címét
- Az első elemet a `head` mutató jelöli ki
- Az utolsó elem nem mutat sehova (`NULL`)

Láncolt lista

- Az üres lista

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



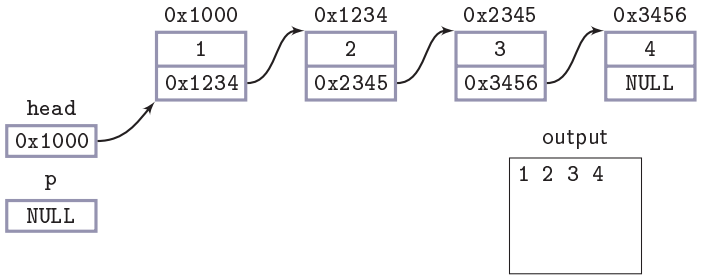
Lista vagy tömb

- A tömb
  - annyi memóriát foglal, amennyi az adatok tárolásához szükséges
  - egybefüggő memóriahelyet 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

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



Lista átadása függvénynek

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

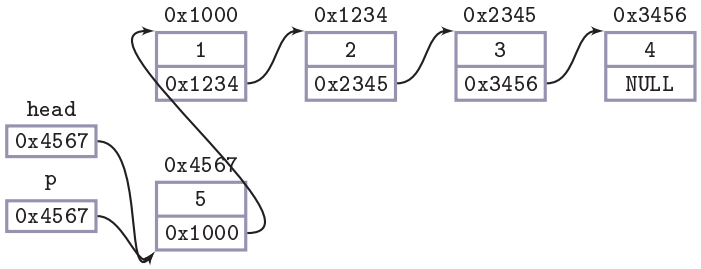
```
void traverse(listelem *head) {
  listelem *p = head;
  while (p != NULL)
  {
    printf("%d ", p->data);
    p = p->next;
  }
}
```

- ugyanaz `for` ciklussal

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


Elem beszúrása lista elejére

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



Elem beszúrása lista elejére függvénnyel

- Mivel beszúráskor a kezdőcím változik, azt vissza kell adnunk

```
listelem *push_front(listelem *head, int d)
{
  listelem *p = (listelem*)malloc(sizeof(listelem));
  p->data = d;
  p->next = head;
  head = p;
  return head;
}
```

- A függvény használata

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

Elem beszúrása lista elejére függvénnyel

- Másik lehetőségként a kezdőcímet cím szerint adjuk át

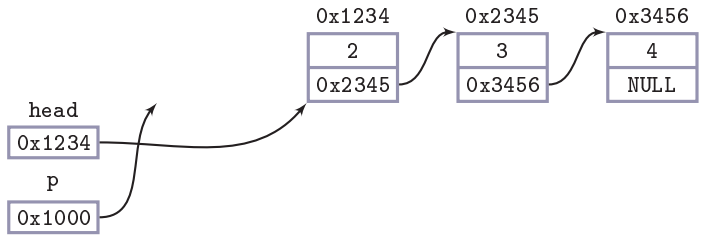
```
1  void push_front(listelem **head, int d)
2  {
3    listelem *p = (listelem*)malloc(sizeof(listelem));
4    p->data = d;
5    p->next = *head;
6    *head = p; /* *head változik, ez nem vész el */
7  }
```

- 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 függvénnyel

```
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 }
```

- Az üres listára külön figyelnünk kell
- Természetesen itt is használhatnánk a `head` címével hívott változatot

Verem

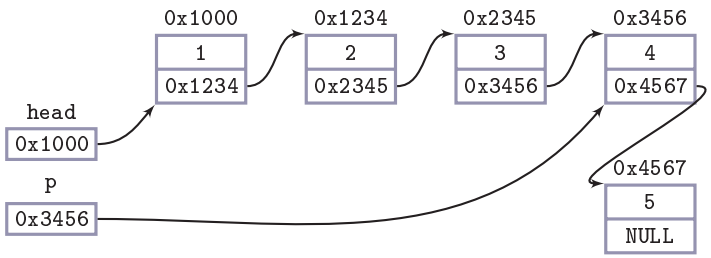
- Ami eddig elkészült, már elég az undo-lista tárolásához

```
1  listelem *head = NULL;            /* üres lista */
2  head = push_front(head, 2);       /* lépés */
3  head = push_front(head, 4);       /* lépés */
4  printf("Az ultoljára 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;
```



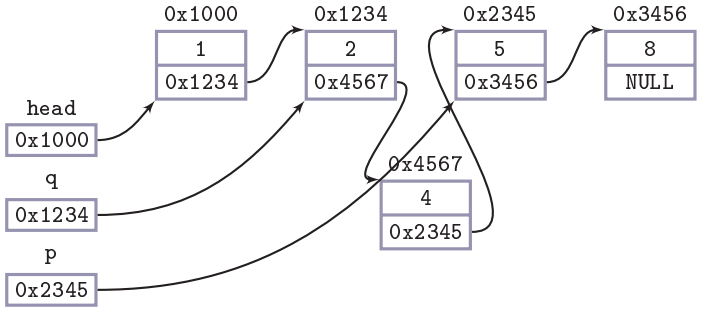
- Üres listára a `p->next != NULL` vizsgálat értelmetlen, azt külön kell kezelnünk!

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 beszúrnunk
- A jelenlegi szerkezetben minden elem csak „maga mögé lát”, nem tudunk elem elé szúrni
- Két mutatóval járjuk be a listát, az egyik mindig eggyel lemarad
- A lemaradó mutató mögé szúrunk

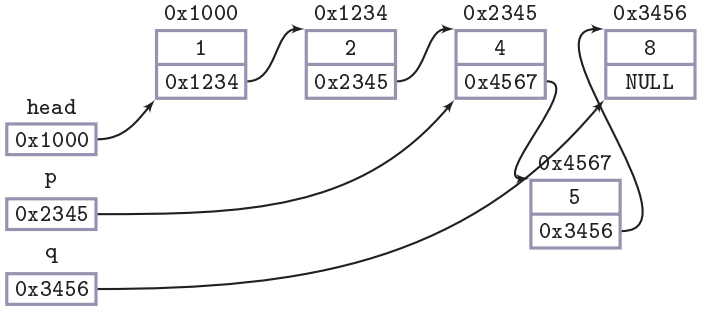
Elem (4) beszúrása rendezett listába

```
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 (4) beszúrása rendezett listába cserével

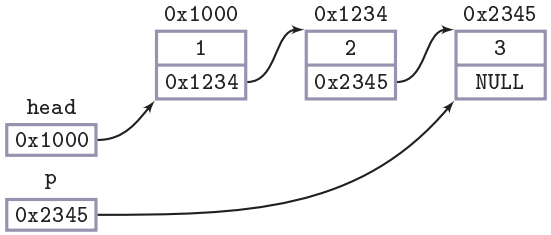
- A lemaradó mutató megspórolható, ha a kiválasztott elem mögé szúrunk, majd cseréljük az adatokat.



- Ez az eljárás csak akkor alkalmazható, ha a már meglévő lista adatait módosíthatjuk, vagyis mások nem hivatkoznak rájuk. Sokszor nem ez a helyzet!

Elem törlése lista végéről

```
p = head;
while (p->next->next != NULL)
  p = p->next;
free(p->next);
p->next = NULL;
```

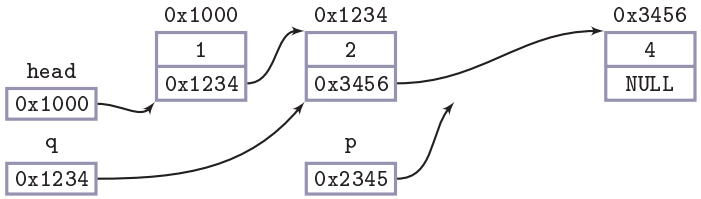


- Ha a lista üres, vagy egy eleme van, a `p->next->next` kifejezés értelmetlen

Adott elem törlése listából

- A `data = 3` elem törlése

```
q = head; p = head->next;
while (p != NULL && p->data != data) {
  q = p; p = p->next;
}
if (p != NULL) { /* megvan */
  q->next = p->next;
  free(p);
}
```



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

Teljes lista törlése

```
void dispose_list(listelem *head)
{
  while (head != NULL)
    head = pop_front(head);
}
```