Programiranje I

Strukture
Unije
Polja bitova
Liste - uvodno

Strukture i pokazivači

- Kao i na svaki drugi podatak, može se deklarisati pokazivač na podatak strukturnog tipa:
  ```
  struct str *s;
  ```
- Preko ovog pokazivača možemo da zauzmemo memoriju:
  ```
  s = (struct str *) malloc(sizeof(struct str));
  ```
- Podacima članovima strukture se, preko pokazivača, može pristupati kao:
  ```
  (*s).i = 4;
  (*s).ch = 'A';
  ```
- Kod ovog načina pristupanja *s je sama struktura.
- Pokazivač na strukturu pokazuje na prvi bajt u memoriji koji je zauzet za podatke te strukture.

Operator -> Struktura u strukturi

- Za pristupanje članovima strukture preko pokazivača, često se koristi operator **->**.

```
s->i = 32;    s->ch = 'A';
```

- Unutar strukture se može naći druga struktura:

```
struct prva {
   int i;
   int b;
};

struct druga {
   struct prva x;
   char c;
};
```

Neka je deklarisana promjenljiva strukturnog tipa:

```
struct druga t;
```

Članovima unutrašnje strukture, tipa prva, se pristupa na sljedeći način:

```
t.x.i = 7;
```

Struktura u strukturi

- Iako dozvoljen i često korišćen koncept struktura u strukturi (eng. nested structure) ne treba koristiti preduboko, odnosno ne bi trebalo da u strukturi koja je članica strukture opet bude struktura i tako u nedogled.
- Strukturu ne moramo imenovati. Ako se ime strukture ne navede, moraju se **odmah** deklarisati promjenljive tog strukturnog tipa:

```
struct {int a; float b;} s1, *s2;
```

- Ovakve neimenovane strukture su najčešće članice drugih struktura.

Struktura u strukturi

- Struktura ne može sadržati istu strukturu za podatak član!

```
struct mojaStruktura {
   int a;
   struct mojaStruktura b;
};
```

- Naime, alokacija promjenljive ovog strukturnog tipa bi podrazumijevala alokaciju cijelog broja i strukture mojaStruktura, u kojoj se nalazi cijeli broj i struktura mojaStruktura, ... Da skratimo, zahtjevala bi se beskonačna alokacija memorije, a to nije dozvoljeno.

Struktura u strukturi

- Također, nije dozvoljeno ni da član strukture bude promjenljiva drugog strukturnog tipa koja se u svojoj realizaciji poziva na prvu strukturu:

```
struct prva {
   struct druga a;
   int b;
};
```

```
struct druga {
   struct prva c;
   int d;
};
```

- Ponovo bi imali pokušaj nedozvoljene beskonačne alokacije memorije.

Pokazivači kao članovi strukture

- Pokazivač na promjenljivu može biti član strukture:

```
struct str {
   int *a;
   char b;
};
```

- Struktura može sadržati i pokazivač na strukturu istog tipa, čime se dobija tzv. **samoreferentna struktura**.

```
struct str {
   struct str *p;
   char b;
};
```

Pokazivač p zauzima memoriju samo za jednu adresu promjenljive strukturnog tipa, pa ne može da dođe do beskonačne alokacije!

Ovo je veoma korisno i upotrebljivo za kreiranje lista i drugih linkovanih tipova podataka.

Referenciranje unaprijed

- Dozvoljeno je deklarisati pokazivač na strukturu prije definicije same strukture:

```
struct str *p;
struct str {
   int a;
   char b;
};
```

- Ova tehnika se naziva referenciranje unaprijed (eng. *forward referencing*).

Unije

- Unija je tip podatka veoma sličan strukturi. Na primjer:

```
union abc {
   int i;
   char c;
} unija;
```

- Na prvi pogled, razlika je u upotrebi ključne riječi union umjesto struct.
- Postoji, ipak, krupna razlika. Unija zauzima memoriju koja je jednaka memoriji potrebnoj za smještaj najvećeg podatka člana (u ovom slučaju najviše memorije je potrebno da bi se smjestio int), a ne koliko je memorije potrebno za smještaj svih promjenljivih članica (kao kod strukture).

Unije

- U primjeru ispod, pojedinačni bajtovi odgovaraju elementima char niza a, dok se sva 4 bajta (pretpostavka je da int zauzima 4 bajta) mogu mijenjati pomoću cjelobrojne promjenljive i.

```
union cde {
   char a[4];
   int i;
};
```

prvi bajt je a[0], …, četvrti a[3], a sva 4 bajta čine i.

- Unije se koriste tamo gdje je potrebno voditi računa o svakom bajtu memorije (obično kod mikrokontrolera).

Polja bitova

- Polja bitova se u C-u deklarišu ključnom riječju struct.

```
struct {
   unsigned int x:1;
   unsigned int y:2;
   unsigned int z:3;
} pb;
```

- Ovim je deklarisana promjenljiva pb koja ima podatke članove x, y i z, koji zauzimaju 1, 2 i 3 **bita**, respektivno, a ne memoriju potrebnu za podatak tipa unsigned int.
- Polje bitova se koristi kod mikrokontrolera i u nekim drugim specifičnim situacijama koje uključuju rad sa registrima procesora. U ovom kursu je od male upotrebne vrijednosti.

Motiv za uvođenje lista

- Više istih tipova podataka se smiješta u niz.
- Raznorodni tipovi podataka se mogu koristiti da modeluju neke pojmove iz realnog svijeta (radnika, studenta, itd.).
- Naravno, postoji i potreba za nizom podataka strukturnog tipa.
- Problem koji postoji je modelovanje **spiska**.
- Pretpostavimo da u firmi imamo radnike sortirane po nekom kriterijumu.
- Kakav problem nam stvara dodavanje novog radnika u spisak?

Spisak – Problem

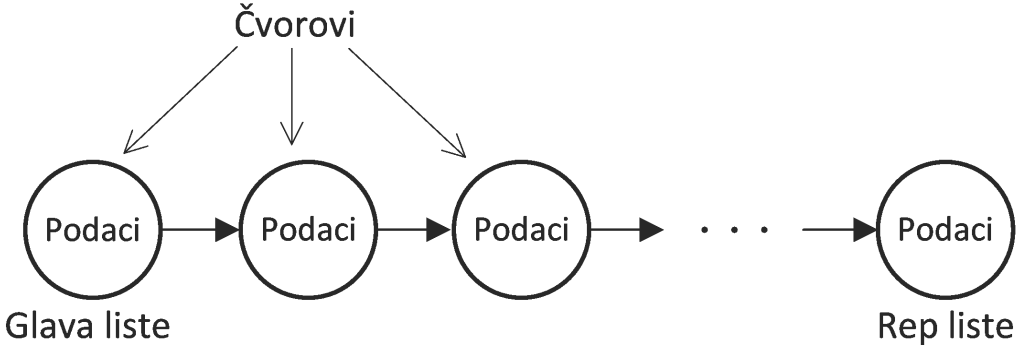
- Postoji više problema kod dodavanja radnika u sortirani spisak.
- Prvi problem je alokacija memorije. Alokacija za prethodni skup radnika je već obavljena, a mi želimo obaviti alokaciju za samo jednog radnika. Biblioteka stdlib.h nudi samo funkciju realloc da realocira kompletnu memoriju za čitav niz. Radnika može biti nekoliko hiljada i sama realokacija memorije može da traje izvjesno vrijeme i da ima neizvjesan ishod.
- Drugi problem je sortiranje radnika. Da li sve sortirati odjednom ili samo dodati novog radnika u već sortirani spisak? Složenost dodavanja se može umanjiti insertion sort algoritmom, ali problem predstavlja to što za svako pomjeranje radnika moramo izvršiti mnoštvo zamjena, a to vodi do operacija sa velikom količinom memorije.

Spisak – Problem

- Sličan problem predstavlja i brisanje radnika iz spiska – opet je potrebno vršiti realokaciju memorije i pomjeranje radnika.
- Bilo kakva reorganizacija spiska podrazumijeva pomjeranje radnika.
- Problemi sa spiskom, a ujedno važnost ovog pojma, su motivisali uvođenje sasvim novog tipa podatka u programersku praksu – **liste**.
- Lista je specijalni tip podatka u kojem svaki član liste (još se zove i **čvor liste**) pamti član koji mu slijedi.

Lista – Osnovni pojmovi

- Prvi čvor liste (koga ne pamti nijedan drugi čvor) se naziva **glavom liste**.
- Čvor liste koji ne pamti nijedan drugi čvor liste naziva se **repom liste**.
- Postoji nekoliko tipova listi. Za sada ćemo se upoznati samo sa **jednostruko povezanom listom**, kod koje svaki čvor, osim poslednjeg, pokazuje samo na naredni čvor liste.



Lista – Realizacija

- U čvorovima liste su upisani neki podaci.
- Podaci su po pravilu strukturnog tipa.
- Ostaje problem da se realizuje metodologija za "pamćenje" narednog elementa liste.
- Postoje dva načina za ovo:
  - preko niza pozicija članova liste i
  - preko samoreferentnih struktura.
- Radi jednostavnosti, obje realizacije ćemo ilustrovati preko liste cijelih brojeva.

Lista preko niza pozicija

- Pretpostavimo da imamo niz u kojem bi brojevi trebali biti sortirani u rastući redosljed. Umjesto da vršimo premještanje članova niza pretpostavimo da je neko već formirao niz indeksa u kom se pamte indeksi narednih elemenata niza. Primjer:

Niz	5	7	1	6	3	2
Indeksi	3	-1	5	1	0	4

glava liste od koje sve počinje

Glava liste preko svoje pozicije pamti da je naredni element sa indeksom 5 - to je broj 2, taj element pamti da je naredni sa indeksom 4 - to je broj 3, ovaj element pamti da je naredni sa indeksom 0 - to je broj 5, ovaj pamti narednog sa indeksom 4, a ovaj narednog sa indeksom 2. Konačno, element sa indeksom 2 je rep liste, koji sa -1 ukazuje da nema narednog elementa.

Dodavanje elementa u listu

- Pošto indeksiranje u C-u započinje sa 0, to znači da rep liste treba da pokaže na, recimo, –1. U prethodnu listu sad treba dodati broj 4.
- Algoritam polazi od glave. Kako je novi element veći od onoga koji se nalazi "u glavi", to znači da se dati element mora postaviti nakon njega. Gleda se, dakle, naredni element u listi. Njegov indeks je sačuvan na poziciji upisanoj u članu niza ispod glave. Element na toj poziciji je 2, a to znači da novododati element treba dalje pomjerati. Naredni u listi je 3, a to znači da idemo dalje. Kako je nakon 3 broj 5, to znači da između 3 i 5 treba dodati novi element (broj 4).

Dodavanje elementa u listu

Polazna lista

Niz	5	7	1	6	3	2
Indeksi	3	-1	5	1	0	4

Novodobijena lista

Niz	5	7	1	6	3	2	4
Indeksi	3	-1	5	1	6	4	0

U prethodnom stanju liste samo je promjenjena pozicija narednog elementa nakon elementa 3. Kod novododatog elementa 4, naredni element je na poziciji gdje je u prethodnom slučaju pokazivao element 3.

Lista preko niza pozicija

- Za memorisanje liste preko nizova je, pored niza podataka, potrebno pamtiti dodatni niz indeksa, istih dimenzija.
- Na prvi pogled, ovo predstavlja problem (više nego duplo memorijskog prostora u odnosu na slučaj prostog sortiranog niza).
- Međutim, kako su po pravilu elementi liste složeniji podaci (recimo, strukture) dodatak od jednog niza indeksa cijelih brojeva nije prevelik.
- Osnovni problem zbog kojeg se liste zapisane preko nizova ne koriste često je činjenica da su elementi ove liste ipak nizovi čije dimenzije na početku treba maksimizovati ili realocirati svaki put kada se dodaje novi član liste.
- Stoga se morao osmisliti drugačiji način za memorisanje liste.

Lista preko samoreferentne strukture

- Mnogo češće se liste kreiraju preko strukture koja za član ima pokazivač na strukturu istog tipa, tj. preko samoreferentne strukture.
- Naša struktura će pokazivati na naredni element u listi ako ima tog elementa, odnosno na NULL ako ga nema (ako je u pitanju rep).
- Da konkretizujemo, posmatraćemo strukturu koju ćemo zvati **lista** i koja će sadržati samo jednu cjelobrojnu promjenljivu (u praksi obično sadrži mnogo toga) i pokazivač na naredni element liste.

Struktura lista

- Deklaracija ovakve strukture:

```
struct lista {
   int i;
   struct lista *next;    //pokazivač na naredni element
};
```

- Listu ćemo formirati na dinamički način - na početku ne postoji nijedan element liste, elemente kreiramo jedan za drugim koristeći dinamičku alokaciju memorije.
- Za dinamičku alokaciju, potrebno je da deklarišemo pokazivač

```
struct lista *p;
```

- Programi koji rade sa listama moraju da pamte pokazivač na glavu liste!

Alokacija glave liste

- Alokacija glave liste se vrši na jednostavan način:

```
p = (struct lista *) malloc(sizeof(struct lista));
                                  // provjeriti da li je alokacija uspjela
p->i = 4;                         // upis broja u listu preko pokazivača
p->next = NULL;                   // sada je p glava, a ujedno i rep liste,
                                  // jer postoji samo jedan čvor liste
```

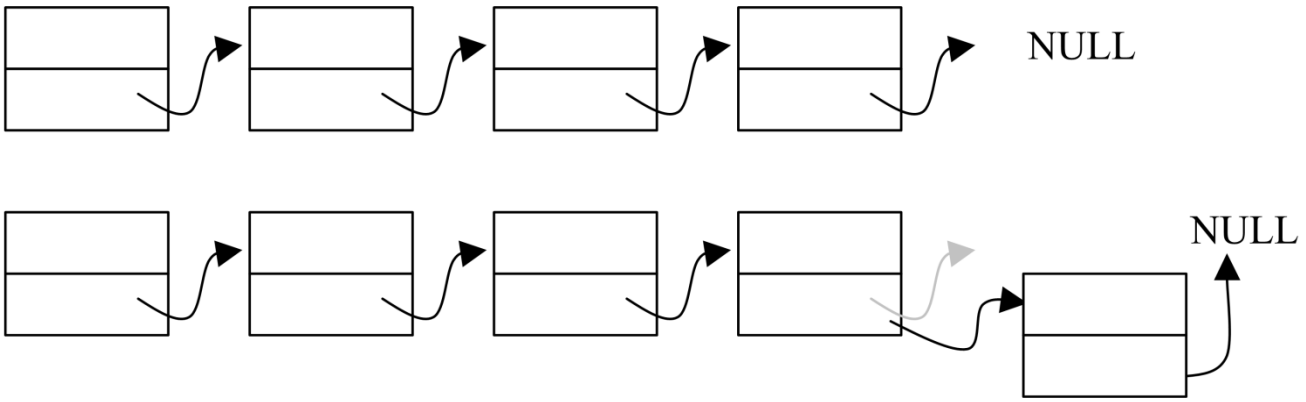
- Pošto moramo pamtiti pokazivač liste, uradimo sljedeće:

```
struct lista *glava = p;
```

- Postoje standardni problemi koje kod listi treba riješiti: dodati rep liste, izbrisati rep liste, dodati element u unutrašnjost liste, izbrisati element iz unutrašnjosti liste, spojiti dvije liste.

Dodavanje elementa na kraj liste

- Pretpostavimo da imamo nepraznu listu prikazanu na slici. Polazimo od glave liste. Ako taj element nije rep (što se provjerava poređenjem pokazivača next tog elementu sa NULL), posmatramo naredni element liste. Proceduru ponavljamo dok ne dođemo do repa. Tada alociramo memoriju za novi element liste (ako to ranije nije urađeno), upišemo podatke u taj element liste i naredimo repu iz prethodne liste da pokaže na novi element. Novi element (tj. novi rep) treba da pokaže na NULL.



Dodavanje elementa na kraj liste

- Prikažimo sada programsku realizaciju (jednu moguću):

```
void dodajRep(struct lista *glava, int k)
{
      struct lista *q;
      while(glava->next != NULL)
            glava = glava->next;
      q = (struct lista *)malloc(sizeof(struct lista));
      if(q == NULL)
            exit(1);
      glava->next = q;
      q->next = NULL;
      q->i = k;
}
```

Funkcija se poziva sa argumentom glavom i sa cijelim brojem koji treba da bude upisan u novi rep liste koji još uvijek ne postoji.

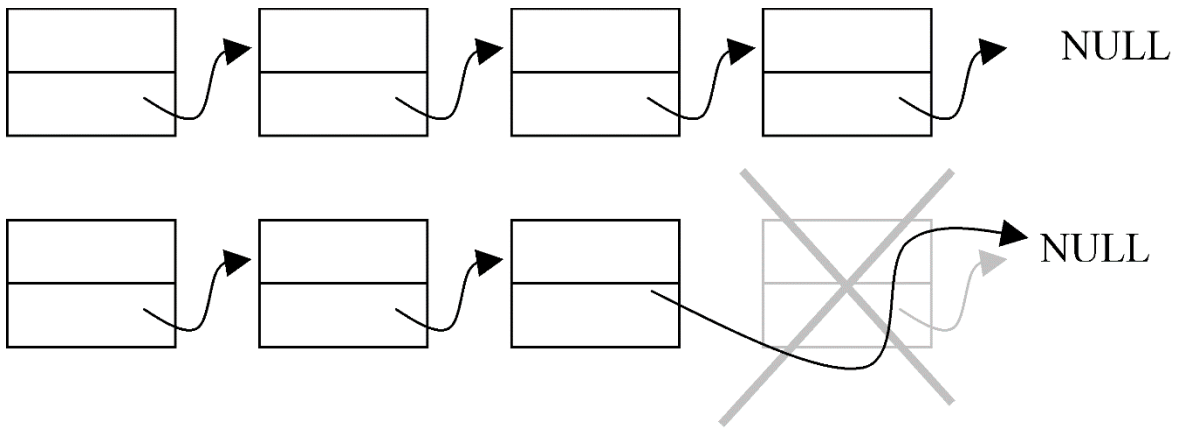
Sve dok ne dođemo do repa liste, krećimo se unapred.

Alokacija memorije za čvor koji će biti novi rep.

Prvo rep pokaže na novi element (čime je postao dio liste), zatim taj element pokaže na NULL (to je novi rep) i na kraju upiši cijeli broj u novi rep.

Brisanje repa liste

- Neka je naš zadatak da napišemo funkciju koja briše posljednji element liste (rep). Moguće su dvije situacije. Jedna je da lista ima više od jednog elementa. Ta funkcija vraća pokazivač na glavu (staru glavu koja se nije promijenila). Moguća je i situacija da lista ima samo glavu koju, naravno, treba izbrisati, a tada funkcija vraća NULL pokazivač.



Brisanje repa liste

```
struct lista *brisiRep(struct lista *glava)
{
    struct lista *p = glava;
    if(glava->next == NULL)
    {
        free(glava);
        return NULL;
    }
    while(p->next->next != NULL)
        p = p->next;
    free(p->next);
    p->next = NULL;
    return glava;
}
```

Glava je ujedno i rep (specijalan slučaj) – dealociramo je i vraćamo NULL.

Dolazimo do pretposljednjeg elementa liste, tj. nakon ove petlje će **p** pokazivati na element liste ispred repa liste.

Oslobađamo memoriju za rep liste, a pretposljednji element postaje rep liste.

Funkcija vraća pokazivač na listu, koja je će biti ista, osim u slučaju da početna lista ima samo jedan čvor, kad se vraća NULL (gornji slučaj).