Programiranje I

Strukture
Unije
Polja bitova
Liste - uvodno

Nizovi i strukture

- Na prethodnom času smo vidjeli da struktura može da ima podatak član koji je niz, ali da nizu moraju biti poznate sve dimenzije.
- Slično, strukture mogu činiti niz.
- Danas ćemo se upoznati sa činjenicom da struktura može da sadrži pokazivače, pa i druge strukture, kao i sa nekim ograničenjima u ovom pravcu.

Strukture i pokazivači

- Kao i na svaki drugi podatak može da se definiše i pokazivač na podatak strukturnog tipa.
  struct str *s;
- Ovo dalje znači da preko ovog pokazivača možemo da zauzmemo memoriju:
  s = (struct str *) malloc(sizeof(struct str));
- Ako je struktura zadata preko pokazivača onda se podacima članovima strukture može pristupati kao:
  (*s).i = 4; ili (*s).ch = 'A';
- Kod ovog načina pristupanja *s je sama struktura.
- Zapamtite da pokazivač pokazuje na prvi bajt u memoriji koji je zauzet za podatake strukturnog tipa na koji pokazuje dati pokazivač.

Strukture u strukturama

- Češće se za pristupanje članovima strukture preko pokazivača na strukturu koristi operator **->**.

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

- Unutar strukture se može nalaziti druga struktura:

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

struct str {
   struct abc x;
   char c;
};
```

Neka je deklarisana promjenljiva strukturnog tipa:

```
struct str proba;
```

Pristup članovima unutrašnje strukture se obavlja na sljedeći način:

proba.x.i=7;

Struktura u strukturi

- Iako dozvoljen i često korišćen koncept struktura u strukturi (nested structure u engleskoj terminologiji) ne treba koristiti "preduboko", odnosno ne bi trebalo da u strukturi koja je članica strukture opet bude struktura i tako u nedogled.
- Strukturu nije neophodno deklarisati sa imenom. Ako se ime strukture ne zadaje odmah moraju se deklarisati promjenljive tog strukturnog tipa:

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

- Ovakve neimenovane strukture su najčešće članovi drugih struktura.

Struktura u strukturi

- Unutar strukture se ne može deklarisati struktura istog tipa!

```
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

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

```
struct A{
   struct B b;
   int c;
};
```

```
struct B{
   struct A d;
   int f;
};
```

- Ponovo bi imali pokušaj nedozvoljene beskonačne alokacije memorije.

Pokazivači - članovi strukture

- Pokazivač na promjenljivu može biti član strukture:

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

- Pored toga, član strukture može biti **pokazivač na strukturu istog tipa**.

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

Ovakav pokazivač zauzima memoriju samo za jednu adresu promjenljive istog tipa, pa ne može da dođe do beskonačne alokacije!!!
Ovo je veoma korisno i upotrebljivo za kreiranje lista i drugih linkovanih tipova podataka.

Unije

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

```
union abc {
   int i;
   char c;
} un1;
```

- 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 narednom primjeru, pojedinačni bajtovi se mogu mijenjati pomoću char niza, dok se sva 4 bajta (pretpostavka je da int zauzima 4 bajta) mogu mijenjati pomoću cjelobrojne promjenljive.

```
union hm {
   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;
} st1;
```

- Na ovaj način je deklarisana promjenljiva st1 koja ima podatke članove x, y i z, ali koji zauzimaju **1**, **2** i **3** **bita**, respektivno, a ne 3 puta memorija za unsigned int podatak.
- Polje bitova se takođe koristi kod mikrokontrolera i u nekim drugim specifičnim situacijama koje uključuju rad sa registrima procesora. Za nas je od relativno 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 nam problem stvara dodavanje novog radnika u spisak?

Spisak - Problem

- Postoji više problema kod dodavanja radnika u sortirani spisak.
- Prvi je vezan za alokaciju memorije. Alokacija za prethodni skup radnika je već obavljena, a mi želimo obaviti alokaciju za samo jednog radnika, ali nam **alloc.h** nudi samo naredbu **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 kako sortirati. 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 operacije sa velikom količinom memorije.

Spisak - Problem

- Sličan problem predstavlja i brisanje radnika iz spiska, kao i druge operacije koje se sa spiskom mogu vršiti.
- Problemi sa spiskom, a ujedno važnost ovog pojma, su motivisali uvođenje sasvim novog tipa promjenljive u programersku praksu.
- Taj tip podatka se naziva **listom**.
- Lista je specijalni tip podatka u kojem svaki član liste (često se naziva **čvorom** ili **vrhom**) pamti **(na neki način)** 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 **(mi ćemo iz razloga koji će biti kasnije objašnjeni umjesto "pamti" govoriti "pokazuje")** nijedan drugi čvor liste naziva se **repom liste**.
- Postoji mnoštvo tipova liste. Za sada ćemo se upoznati samo sa **jednostruko povezanom listom**, kod koje svaki čvor, osim poslednjeg, pokazuje samo na jedan čvor, tj. naredni element 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, obije realizacije ćemo ilustrovati preko liste cijelih brojeva.

Lista sa nizom 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 pozicija u kojem se pamte naredni elementi niza. Primjer:

Niz	5	7	**1**	6	3	2
Pozicije	4	0	**6**	2	1	5

glava liste od koje sve počinje

Glava liste preko svoje pozicije pamti da je naredni element na poziciji 6 - to je broj 2, taj element pamti da je naredni na poziciji 5 - to je broj 3, ovaj element pamti da je naredni na poziciji 1 - to je broj 5, ovaj pamti narednog na poziciji 4, a ovaj narednog na poziciji 2 i, konačno, element na poziciji 2 je rep liste, koji sa 0 indicira da nema narednog elementa.

Dodavanje elementa u listu

- Napominjemo da indeksiranje u C-u započinje sa indeksom 0, a to znači da npr. rep liste treba da pokaže na, recimo, –1.
- Pretpostavimo da u prethodnu listu 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 elemenat mora postaviti nakon njega; gleda se, zatim, 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 novododati element treba da ga prati. Kako je nakon 3 element 5 to znači da između 3 i 5 treba dodati novi element.

Dodavanje elementa u listu

Niz	5	7	**1**	6	3	2
Pozicije	4	0	**6**	2	1	5

polazna lista

Niz	5	7	**1**	6	3	2	4
Pozicije	4	0	**6**	2	7	5	1

novodobijena lista

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

Zapis liste preko nizova

- Za memorisanje liste preko nizova je, pored niza koji predstavlja podatke, potrebno pamtiti dodatni niz indeksa, istih dimenzija, a uvijek je potrebno pamtiti i poziciju glave liste.
- Na prvi pogled, ovo je problem (više nego duplo memorijskog prostora u odnosu na slučaj prostog sortiranog niza).
- Međutim, kako su po pravilu elementi liste složeniji podaci (recimo strukture) onda dodatak od jednog niza indeksa cijelih brojeva nije prevelik.

Zapis liste preko nizova

- 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.
- Prije nego pređete na tu metodologiju pokušajte da za listu zapisanu preko niza sa indeksima osmislite sljedeće algoritme i napišete programe koji ih rješavaju:
  brisanje elementa iz sredine liste, dodavanje elementa na kraj liste, brisanje elementa sa kraja liste.

Lista = Samoreferentna struktura

- Mnogo češće se liste kreiraju preko strukture čiji je jedan član pokazivač na strukturu istog tipa.
- Kod diskusije vezane za strukture vidjeli smo da je ovo dozvoljeno.
- 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 pojednostavimo, posmatraćemo strukturu koju ćemo zvati **lista** (naravno, ovaj naziv nije obavezan) 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 (koja se naziva samoreferentom):

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

- Smatraćemo da na početku ne postoji nijedan element liste. To znači da u glavnom programu nije deklarisana promjenljiva tipa lista, već samo pokazivač na promjenljivu tog tipa (obično se deklariše nekoliko takvih pokazivača):
  `struct lista *gl;`
- Programi koji rade sa listama moraju da budu realizovani tako da jedan pokazivač uvijek pokazuje na glavu (početak) liste.

Alokacija glave liste

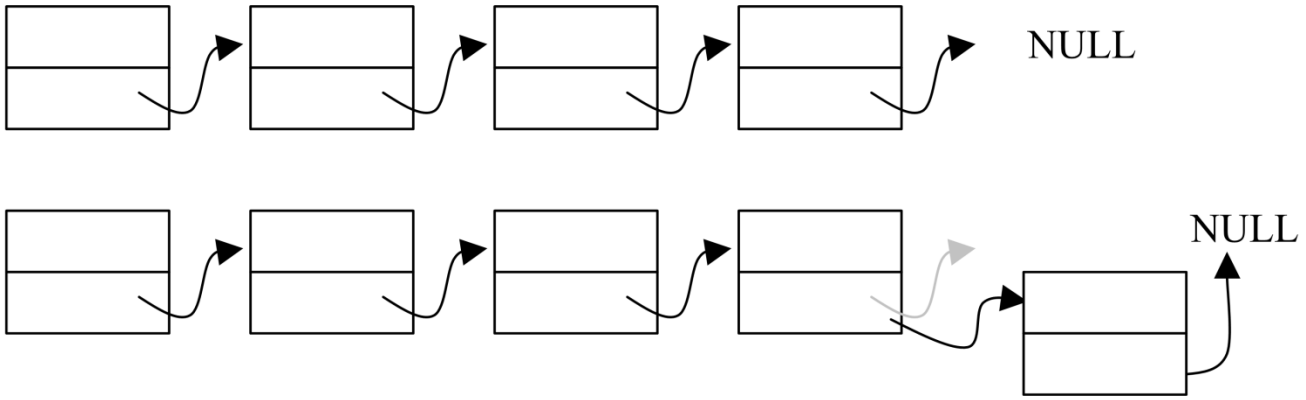
- Alokacija glave liste se vrši na jednostavan način:

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

- Postoje standardni problemi koje kod listi treba riješiti. Na primjer, dodati rep liste, izbrisati rep liste, dodati element u unutrašnjost liste i izbrisati element iz unutrašnjosti liste, spojiti dvije liste.
- Kod listi koje su zadate preko samoreferentnih struktura, ovo se može realizovati koristeći rekurzivne funkcije.

Dodavanje elementa na kraj liste

- Pretpostavimo da imamo nepraznu listu prikazanu na slici. Polazimo od glave liste. Ako taj element nije rep (a to se provjerava poređenjem pokazivača next upisanog u tom 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 (odnosno 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.

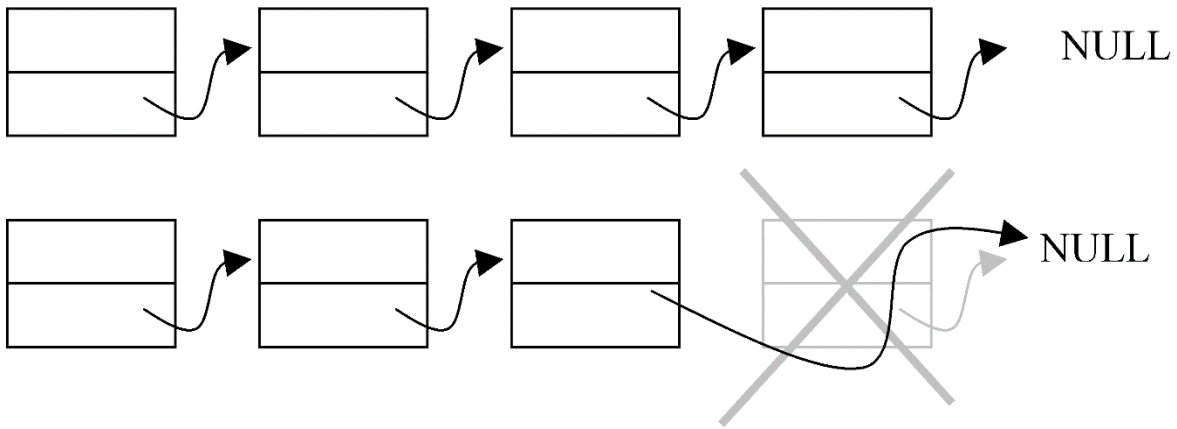
Sve dok ne dođemo do repa liste, krećimo se unapred.

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 (ponaša se kao i svaki drugi rep) i na kraju se u njega upiše cijeli broj.

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 promjenila). 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).