Algoritmi i strukture podataka

Tutorial o stogovima, listama i sortovima

autor: Marko Čupić

C TIPS: STOG

Stog je struktura koja obično služi za pohranu podataka. Ideja je slijedeća: gurnete jedan objekt na stog, nešto napravite, gurnete drugi objekt na stog, opet nešto radite, uzmete natrag drugi objekt sa stoga, uzmete natrag prvi objekt sa stoga, pa opet nešto stavite na stog, itd. Vjerojatno ste primjetili da objekte vadite sa stoga u suprotnom redoslijedu od onoga kojim su oni stavljani na stog. Stručno govoreći, mehanizam se zove LIFO ("Last in, first out", odnosno u prevodu "zadnji unutra, prvi van"), ili pak FILO (samo okrenite, pa ste opet na istom), važno je samo da termine ne pobrkate sa FIFO ("First in, first out", "prvi unutra, prvi van"), ili pak LILO (jako rijetko, ali znači isto što i prethodno, "Last in, Last out", tj. zadnji unutra, zadnji van). Stog se može iskoristiti na najrazličitije načine. Zamislite samo ovo: imate polje brojeva koje želite sortirati. Nadite u polju najveći i gurnite ga na stog (i maknite iz polja); ponavljajte ovo tako dugo dok ima brojeva u polju. Kada polje ostane prazno, povadite onoliko elemenata sa stoga koliko je elemenata bilo polju i vraćajte ih u polje od prve pozicije na dalje. Što ste dobili? Sortirano polje! Naravno, ima daleko boljih i bržih algoritama za sortiranja, ali tek toliko da spomenem da se i to može stogom. Nadalje, stog je vrlo koristan kod rekurzija, a pogotovo kada rekurzivne funkcije pretvarate u nerekurzivne. Tada se obavezno koristi stog.

IZVEDBE STOGA: POLJE

Jedan od čestih načina kako napraviti stog je pomoću polja. Polje se puni od početka prema kraju, a elementi se vade od kraja prema početku. Za demonstraciju ćemo ilustrirati stog integera. Evo kako bi to izgledalo u C-u:

```
int DodajNaStog( int *stog, int *vrh, int max, int podatak ) {
  (*vrh)++;
  if( *vrh >= max ) { (*vrh)--; return 1; }
  stog[*vrh] = podatak;
  return 0;
}

int SkiniSaStoga( int *stog, int *vrh, int *podatak ) {
  if( *vrh < 0 ) return 1;
  *podatak = stog[*vrh];
  (*vrh)--;
  return 0;
}

void main( void ) {
  int stog[20], vrh, max;
  int a,b,c,t;

  max = 20;
  vrh = -1;

  a = 10; b = 15; c = 20;

  printf( "Dodajem %d\n", a );
  DodajNaStog( stog, &vrh, max, a );
  printf( "Dodajem %d\n", b );
  DodajNaStog( stog, &vrh, max, b );
  printf( "Dodajem %d\n", c );
  DodajNaStog( stog, &vrh, max, c );

  SkiniSaStoga( stog, &vrh, &t );
  printf( "Skidam %d\n", t );
  SkiniSaStoga( stog, &vrh, &t );
  printf( "Skidam %d\n", t );
  SkiniSaStoga( stog, &vrh, &t );
  printf( "Skidam %d\n", t );
}
```

Prvi parametar obje funkcije je polje koje glumi stog. Parametri koji se predaju uz ovaj su i pozicija zadnjeg elementa koji je upisan na stog, te kapacitet stoga. Funkcije su izvedene tako da vračaju pokazatelj uspješnosti izvođenja tražene operacije: 0 ako je operacija uspjela, odnosno 1 ako je došlo do greške. Pogledajmo kako radi funkcija `DodajNaStog`. Pokazivač vrha stoga povečava se za jedan. Zatim provjeravamo jesmo li prekoračili kapacitet polja. Ako jesmo, vračamo pokazivač vrha stoga na prethodnu vrijednost i prekidamo funkciju sa porukom greške. Inače na stog pohranjujemo dani podatak i vračamo nulu. Funkcija `SkiniSaStoga` prvo provjerava ima li uopće elemenata na stogu. Ako nema, vrača se poruka greške. Inače se podatak kopira u predanu varijablu, pokazivač vrha stoga se smanjuje za jedan i funkcija vrača nulu kao znak da je podatak uspješno pročitan. U funkciji main krećemo sa deklaracijom polja za stog te varijablama za čuvanje vrha stog i njegove veličine. Zatim deklariramo još tri integera koja ćemo pohraniti na stog, i konačno jedan integer u kojeg ćemo spremati vrijednosti sa stoga. Kapacitet stoga postavljamo na 20, vrh na -1 jer je stog prazan. Funkcijama `DodajNaStog` dodajemo sva tri elementa na stog, a funkcijama `SkiniSaStoga` vrijednosti skidamo sa stoga i ispisujemo na ekran. Na ekranu se dobije (prema očekivanjima):

```
Dodajem 10
Dodajem 15
Dodajem 20
Skidam 20
Skidam 15
Skidam 10
```

Koristite li u svom programu često ovu strukturu, moram vas upozoriti na dva ozbiljna nedostatka. Prvi vrijedi općenito. Pogledajte deklaracije funkcija za dodavanje i skidanje sa stoga. Prenosi se jako puno suvišnih parametara što usporava program. Ovo treba izbjeći. U nastavku ću pokazati kako. Drugo upozorenje vrijedi ako umjesto integera na stog stavljate veće objekte, poput velikih struktura (typedef struct). Primjetite da funkcija za dodavanje na stog predaje objekt *by value*. To znači da funkcija prima kopiju cijelog objekta, što je opet loše po vaše performanse. Zato funkciju za dodavanje treba preurediti tako da prima samo adresu onoga što će pohraniti na stog.

IZVEDBE STOGA: LISTA

Eh, da. Još jednom sve iz početka. Prvo, ako ne razumijete liste, pročitajte si tekst **C Tips: Liste**. Onda nastavite čitati dalje. Ja ću u nastavku za ostvarivanje stoga koristiti slijedeću strukturu člana liste:

```
typedef struct SCvorStogListe {
  struct SCvorStogListe *Prethodni;
  char x;
} StogLista;
```

Kako vidite, koristiti ćemo jednostruko povezanu listu, ali povezanu unatrag. Razlog ovome je taj što implementacija funkcija za dodavanje i skidanje na stog ne zahtjeva slijedeći čvor, već samo prethodni. Stoga je razumno ono što ne koristimo izbaciti van! Osim ovoga uočite i član nazvan **x**, tipa char, odnosno zauzeća 1 bajt. Čemu on služi? Pa lista osim svog uređenja treba sadržavati i nekakve podatke. Na početku strukture nalazi se pokazivač na prethodni čvor. Iza toga slijedi mjesto za podatke. Prvi bajt tih podataka upravo je x. I ne dajte se prevariti. Nećemo pamtiti samo jedan podatak. Uzmimo npr. da želimo pohraniti jedan double. Tada bi umjesto char x trebalo stajati double x. Zatim bismo alocirali jednu takvu strukturu i dodali je u listu. No ako piše char x, kao kod nas, tada je očito da nam je struktura premala. Koliko? Pa sizeof(double)-sizeof(char) tj. 8-1 što je 7 bajtova manjka. Da li je to problem? Pa zapravo i nije. Kada ćemo alocirati strukturu, umjesto `malloc(sizeof(StogLista))` koristiti ćemo `malloc(sizeof(StogLista)+sizeof(double)-1)`. Ovime smo dobili strukturu koja je upravo dovoljno velika da zapamti naš double! Kako ćemo ga pohraniti? Pa početak mu je tamo gdje se nalazi x. Zbog toga sve što trebamo učiniti je uzeti adresu od x, ukalupiti je u pointer na double (što možemo jer smo si alocirali dovoljno memorije u strukturi!), i zatim preko pointera dodijeliti vrijednost! Zbog toga što funkcije pišemo tako da rade sa proizvoljnim veličinama objekata, umjesto dodjeljivanja koristiti ćemo funkciju za kopiranje memorije memcpy. Nadalje, da bi funkcije znale kako dodavati i skidati sa stoga, potreban im je pokazivač na listu, te informacija o veličini objekta. To su već dva podatka. Poučeni

iskustvom iz prethodno opisane izvedbe stoga preko polja, sve te podatke potrpati ćemo u jednu strukturu pa funkcijama za dodavanje i skidanje predavati samo pointer na tu strukturu u kojoj su sadržani svi relevantni podaci za rad obiju funkcija. Strukturu ćemo nazvati TDStog (D od dinamičkog, da ga ne miješamo sa gornjom deklaracijom statičkog stoga). I opet ćemo napisati i funkcije za kreiranje te strukture te za brisanje te strukture. Pogledajmo prvo kako izgleda deklaracija same TDStog strukture te funkcija za njezino kreiranje i brisanje:

```
typedef struct {
  StogLista *stog;  // lista
  int objvel;       // velicina objekta
} TDStog;

TDStog *KreirajDinStog( int velicina_objekta ) {
  TDStog *stog;

  stog = malloc( sizeof(TDStog) );
  if( stog == NULL ) return NULL;
  stog->stog = NULL;
  stog->objvel = velicina_objekta;
  return stog;
}

void OslobodiDinStog( TDStog *stog ) {
 void *podatak;

 if( stog == NULL ) return;

 if( stog->stog != NULL ) {
  podatak = malloc( stog->objvel);
  while(!SkiniSaDinStog(stog,podatak));
 }
 free(stog);
}
```

Kako vidite, struktura `TDStog` pamti samo pokazivač na listu i veličinu objekta. Podatak o maksimalnom kapacitetu i trenutnom vrhu stoga nema potrebe pamtiti eksplicitno preko još dva parametra. Razlog tomu je da kapacitet stoga ovisi samo o količini slobodne memorije pa ne možemo fiksno odrediti kapacitet, a i ne želimo fiksan kapacitet! S sdruge strane, podatak o vrhu stoga pamti se preko same liste jer ona uvjek pokazuje na vrh stoga, a ne na prvi alocirani element! Zato i koristimo umjesto Slijedeći član pokazivač Prethodni član da bismo znali ići prema početku. Funkcija `KreirajDinStog` alocira mjesto za jednu `TDStog` strukturu, te inicijalizira listu na praznu, i veličinu objekta na predanu veličinu. Ukoliko je alokacija uspjela, vrača se pokazivač na strukturu, inaće se vrača NULL. Funkcija za oslobađanje stoga `OslobodiDinStog` prije samog oslobađanja strukture `TDStog` mora provjeriti je li lista prazna. Ako nije, treba i nju osloboditi. Za oslobađanje liste kreira se mjesto za jedan objekt stoga, i poziva se funkcija `SkiniSaDinStog` tako dugo dok ima elemenata na stogu. Onog trena kada nestane elemenata, znači da je i lista nestala! Na taj način oslobođena je lista, pa se konačno oslobađa i TDStog struktura. I time je funkcija gotova. Sada ćemo pokazati kako trebaju izgledati funkcije za dodavanje na stog i skidanje sa stoga:

```
int DodajNaDinStog( TDStog *stog, void *Podatak ) {
  StogLista *Element;

  Element = malloc(sizeof(StogLista)-1+stog->objvel);
  if( Element == NULL ) return 1;
  memcpy( &Element->x, Podatak, stog->objvel );
  DodajUListu( &stog->stog, Element );
  return 0;
}
```

```
int SkiniSaDinStog( TDStog *stog, void *Podatak ) {
  StogLista *Element;

  Element = SkiniIzListe( &stog->stog );
  if( Element == NULL ) return 1;
  memcpy( Podatak, &Element->x, stog->objvel );
  free(Element);
  return 0;
}
```

Funkcija `DodajNaDinStog` prvo alocira jedan čvor za listu. Uočite već prije opisan način izračuna potrebne memorije za strukturu `malloc(sizeof(StogLista)-1+stog->objvel)`. Provjeravamo je li alokacija uspjela i u slučaju neuspjeha vračamo 1 kao znak greške. Inače kopiramo predani podatak na odgovarajuće mjesto u strukturi (sjećate se, prvi bajt počinje na x, pa zato uzimamo adresu od x i kopiramo od te adrese na dalje). Konačno tako dobiveni čvor liste dodajemo u listu pozivom funkcije `DodajUListu` čiju ćemo implementaciju pokazati malo kasnije, te vračamo 0 kao znak uspjeha. Funkcija `SkiniSaDinStog` radi upravo obratnim redoslijedom. Prvo se sa liste uzme zadnji dodani element pozivom `SkiniIzListe` funkcije i provjerava se je li skidanje uspjelo. Ako nije, znači da lista više nema elemenata, tj. da je stog prazan, pa vračamo 1 i završavamo funkciju. Inače, kopiramo podatke natrag u predanu varijablu, oslobađamo memoriju zauzetu čvorom koji je čuvao podatke i vračamo 0 kao znak uspješnosti funkcije. Još nam je ostalo da pogledamo i funkcije za dodavanje u listu i brisanje iz liste.

```
void DodajUListu( StogLista **Lista, StogLista *Clan ) {
  Clan->Prethodni = *Lista;
  (*Lista) = Clan;
}

StogLista *SkiniIzListe( StogLista **Lista ) {
  StogLista *t;

  if( *Lista == NULL ) return NULL;
  t = *Lista;
  *Lista = (*Lista)->Prethodni;
  return t;
}
```

Funkcija `DodajUListu` kreće od jednostavne pretpostavke. Ono što se preda kao pokazivač na listu nije pokazivač na prvi element liste, već je pokazivač na zadnji element liste. Tada je dodavanje u listu jako jednostavno. Čvor koji dodajemo postati će zadnji čvor. Njegov prethodni čvor je onda dosadašnji zadnji čvor liste a to je upravo *Lista. Ovime je zadnji čvor liste i službeno postao cvor Clan. No kako *Lista mora pokazivati na zadnji član (a trenutno pokazuje na predzadnji), vršimo i tu korekciju drugom naredbom: `(*Lista) = Clan`. Ovime smo opet očuvali pretpostavku uređenosti liste: *Lista opet pokazuje na zadnji element! I funkcija za skidanje iz liste `SkiniIzListe` koristi tu preptostavku. Prvo se provjerava je li lista prazna. Ako je, vrača se NULL i izlazi se iz funkcije. Inače, kako je *Lista pokazivač na zadnji član liste, njega upravo trebamo skinuti! Zato ga pospremamo u pomočnu varijablu t, a pokazivač *Lista korigiramo tako da pokazuje ponovno na zadnji element, koji je sada (*Lista)->Prethodni. Uočite da ako je dotični element bio jedini u listi, tada smo ovime pokazivač *Lista automatski postavili na NULL jer je prethodni element prvog elementa očito NULL!

JOŠ MALO PA KRAJ

Ne, neću više ništa novoga reći o stogu. Ali jedna napomena. Budući da je ovaj tekst pisan jako opširno, tekst o strukturi zvanoj red oslanjati će se na dijelove ovoga teksta zbog toga što se red može implementirati na identične načine kao i stog. Razlika je utolika što struktura red radi na principu FIFO (prvi unutra, prvi van) za razliku od stoga koji je primjer LIFO strukture (zadnji unutra, prvi van). Dakle promjeniti će se samo funkcije za ubacivanje i skidanje sa liste (ili polja ako je statička struktura). Funkcija DodajNaStog, SkiniSaStoga, DajPotrebnuVelicinu i sve ostalo ostaje nepromjenjeno!

C Tips: LISTE

Liste kao cjelina vrlo je opširna tema; ima ih svakakvih: jednostruko povezanih, dvostruko povezanih, sa više ključeva koje opet mogu biti jednostruko povezane ili dvostruko povezane, zatim postoje cirkularne liste i da ne nabrajam ostatak. No ograničimo li se na liste sa jednim ključem, ili kraće rečeno obične najčešće korištene liste, lako ćemo objasniti o čemu se tu zapravo radi. U osnovi, lista je niz povezanih elemenata koji se u memoriji ne moraju nalaziti u nizu; suprotno njima, npr. polje je niz elemenata koji se u memoriji nalaze u nizu! Osnovni gradbeni element svake liste je pokazivač na slijedeći element. Uz njega, svaki element posjeduje i dio koji se koristi za svrhe samog programa, tj. mjesto gdje pamtimo nekakve podatke. Pogledajmo npr. kako bismo definirali element liste koji će pamtiti nečije ime i prezime:

```
typedef struct ClanListeSt {
  struct ClanListeSt *Slijedeci;
  char Ime[20];
  char Prezime[30];
} ClanListe;
```

Ovime smo definirali novi tip podatka: ClanListe, koji je zapravo sinonim za tip: struct ClanListeSt. Ovo je struktura koja sadrži tri elementa: pokazivač na isti takav element, te dva proizvoljna podatka: Ime i Prezime, što su polja za pohranu nečijega imena i prezimena. Kada gradimo listu, u nju čemo dodavati elemente koji su upravo ovoga tipa. Pogledajmo kako bi izgledala funkcija koja će ovakav element dodati u listu.

DODAVANJE U LISTU NA TRI NAČINA

```
void DodajUListu( ClanListe **Lista, ClanListe *Clan ) {
  ClanListe *p;

  Clan->Slijedeci = NULL;
  if( *Lista == NULL ) {
    *Lista = Clan;
  } else {
    p  = *ClanListe;
    while( p->Slijedeci != NULL ) p = p->Slijedeci;
    p->Slijedeci = Clan;
  }
}
```

Ova funkcija dodaje element Clan na zadnje mjesto liste. Budući da će Clan biti zadnji, Clan->Slijedeci postavlja se na NULL pokazivač. Funkcija prima dva parametra: pokazivač na pokazivač na listu, tj. na prvi element liste, i pokazivač na element koji se želi dodati u listu. Možda se pitate zašto je prvi parametar dvostruki pokazivač, pa da i to objasnimo. Ako je lista prazna, tj. ima nula članova, tada je pokazivač na listu jednak NULL pokazivaču. No kada dodamo prvi element u listu, očito da pokazivač na listu moramo promjeniti iz NULL pokazivača u pokazivač na taj element; to će biti moguće samo ako znamo adresu pokazivača na listu što je drugim riječima pokazivač na pokazivač na listu! Pogledajmo sada kako radi funkcija.
Prvo se inicijalizira Clan->Slijedeci na NULL. Razlog tomu je činjenica da će taj element kada se jednom doda u listu biti zadnji. Tada je očito da slijedećeg elementa nema. Zatim provjeravamo je li lista prazna. Ako je, Clan koji dodajemo postaje prvi član liste pa zato mjenjamo pokazivač na listu tako da pokazuje na taj element. Ukoliko lista nije prazna, pomočnoj varijabli p dodjeljujemo pokazivac na prvi član liste, te ulazimo u while petlju koja se vrti tako dugo dok p ne postane zadnji element liste. Ovo se postiže jednostavnom provjerom: pitamo se da li postoji p->Slijedeci? Ako postoji, p postaje p->Slijedeci i tako redom sve do trenutka kada ustanovimo da p->Slijedeci ne postoji. Tada iskačemo iz petlje jer je p očito posljednji element liste. Novi član tada dodajemo tako da p->Slijedeci postavimo na Clan. I ovime je postupak dodavanja u listu završen. Pogledajmo kako bismo riješili dodavanje novih elemenata na početak.

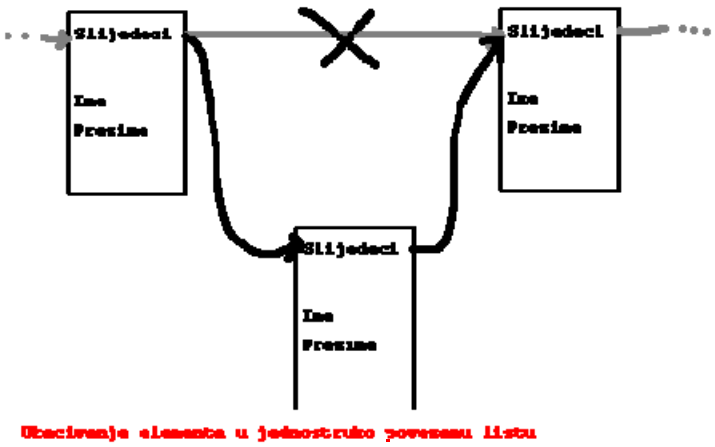
```
void DodajUListu2( ClanListe **Lista, ClanListe *Clan ) {
  if( *Lista == NULL ) {
    Clan->Slijedeci = NULL;
    *Lista = Clan;
  } else {
    Clan->Slijedeci = *Lista;
    *Lista = Clan;
  }
}
```

Što ovdje radimo? Prvo provjerimo je li lista prazna. Ako je, jednostavno dodamo element kao i u prethodnoj funkciji. Međutim, ako lista nije prazna, najprije za slijedeci element novog člana dodjelimo prvi član liste, čime naš novi član tvori listu u kojoj je on prvi. Zatim pokazivač na prvi član liste jednostavno preselimo da pokazuje na naš novi čaln i ovime je dodavanje u listu završeno.
Još nam prostaje da pogledamo kako se dodaju elementi u na neki način uređenu listu. Npr. želimo dodati novi element u listu koja je sortirana po prezimenu ali tako da lista i dalje ostane sortirana.

```
void DodajUListu3( ClanListe **Lista, ClanListe *Clan ) {
  ClanListe *p,*r;
  int i;

  if( *Lista == NULL ) {
    Clan->Slijedeci = NULL;
    *Lista = Clan;
  } else {
    p = *Lista; r = NULL;
    do {
      i = strcmp(Clan->Prezime,p->Prezime);
      if( i <= 0 ) break;
      r = p; p = p->Slijedeci;
    } while( p != NULL );
    if( r == NULL ) {
      Clan->Slijedeci = *Lista;
      *Lista = Clan;
    } else {
      Clan->Slijedeci = r->Slijedeci;
      r->Slijedeci = Clan;
    }
  }
}
```

Možda izgleda malo kompliciranije nego prethodne funkcije za dodavanje, nedajte se zbuniti jer uopće nije. Postupak je slijedeći. Prvo provjerimo je li lista prazna. Ako je, novi Clan jednostavno dodamo tako da pokazivač liste postavimo da pokazuje na taj Clan. Ako lista nije prazna, ulazimo u do-while petlju. Pri tome se koristimo sa dva pomočna pokazivača: p i r, te pri tome vrijedi da r pokazuje na prethodni element od onoga na koji pokazuje p. Tako na početku inicijaliziramo p da pokazuje na prvi član, a r onda mora pokazivati na NULL jer prvi član nema prethodnika. U petlji uspoređujemo Clan->Prezime sa p->Prezime funkcijom strcmp. Ako je rezultat manji ili jednak nula, to znači da Clan->Prezime dolazi prije p->Prezime; sada se sjetimo da r pokazuje na član prije p-a; znači da novi član moramo umetnuti između r-tog i p-tog. I iskačemo iz petlje. Ako je rezult veci od nule, nastavljamo dalje kroz petlju. r postaje p, a p postaje p->Slijedeci. I petlja se vrti sve dok p ne postane NULL kada izlazimo iz petlje. Sada samo treba ubaciti Clan iza r-tog. Ako je r jednak NULL, znači da se član umeće ispred prvoga, dakle on postaje prvi. Inače ga normalno ubacujemo u listu.



Ubacivanje elemenata u jednostruko povezanu listu

PROLAZ KROZ LISTU

Ovo je valjda najlakše od svega. Dovoljan vam je jedan pokazivač.

```
void ProlazKrozListu( ClanListe **Lista ) {
  ClanListe *p;

  p = *Lista;
  while( p != NULL ) {
    // ovdje nešto napraviti, npr. ispis:
    printf("Ime i prezime: %s %s\n",p->Ime,p->Prezime);
    p = p->Slijedeci;
  }
}
```

I odmah jedno upozorenje, sigurno se uočili komad gdje piše "ovdje nešto napraviti". Upozorenje se odnosi na činjenicu da to nešto ne smije mijenjati ono na što p pokazuje (za uređene liste, jer bi se mogla narušiti uređenost), a pogotovo ne smije osloboditi taj clan iz memorije (za sve liste)! Ukoliko želite osloboditi cijelu listu iz memorije, morati ćete posegnuti za tehnikom prolaska sa dva pokazivača. Zašto? Pa ako najprije oslobodite p iz memorije, a zatim napišete p = p->Slijedeci što ste napravili? P je oslobođen pa je p->Slijedeci ilegalan poziv! ZABRANJENO! Iako mislim da sam ovako nešto zapravo i vidio u skriptama! Pravilo glasi: memoriju kojoj niste vlasnik ne dirajete! A kako ste oslobodili ono na šta pokazuje p, više ne smijete koristiti p->Slijedeci kao ni p->Ime ni išta drugo! Zato prije oslobađanja treba zapamtiti što je bilo iza p! Evo kako:

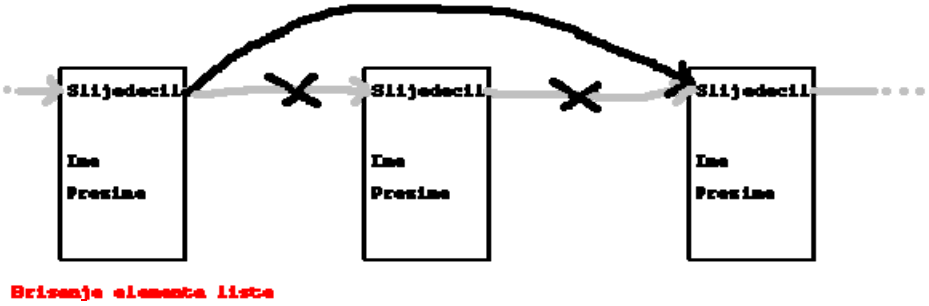
```
void ProlazKrozListu2( ClanListe **Lista ) {
  ClanListe *p, *r;

  p = *Lista;
  while( p != NULL ) {
    r = p->Slijedeci;
    // ovdje nešto napraviti, npr. oslobodi clan:
    free(p);
    p = r;
  }
}
```

Ovo sada funkcionira savršeno. I ono na što morate obratiti pažnju je da nakon izvršenja gornje funkcije lista više ne postoji, iako *Lista još uvjek pokazuje na komad memorije koji sada više nije u našem vlasništvu. Dakle najispravnije bi bilo kao zadnju naredbu funkcije napisati *Lista = NULL;

BRISANJE ELEMENATA LISTE

Odmah da razjasnimo nešto. Kada kažem brisanje, zapravo mislim na izbacivanje elementa iz liste, ali ne i na njegovo oslobađanje iz memorije. Napisati ćemo funkciju za brisanje kojoj ćemo predati pokazivač na listu, i pokazivač na član koji želimo obrisati.



Brisanje elemenata liste

```
void ObrisiIzListe( ClanListe **Lista, ClanListe *Clan ) {
  ClanListe *p,*r;

  if( *Lista == NULL ) {
    return;
  } else {
    p = *Lista; r = NULL;
    do {
      if( p == Clan ) break;
      r = p; p = p->Slijedeci;
    } while( p != NULL );
    if( p == NULL ) return;
    if( r == NULL ) {
      *Lista = Clan->Slijedeci;
    } else {
      r->Slijedeci = Clan->Slijedeci;
    }
  }
}
```

I opet ćemo analizirati rad funkcije. Prvo se provjerava je li lista prazna. Ako je, nemamo što brisati. Inače, opet koristimo dva pokazivača, p i r, pri čemu r pokazuje na član ispred onoga na koji pokazuje p. Ulazimo u do-while petlju koja se vrti tako dugo dok p ne postane NULL pokazivač čime indicira da se traženi element uopće ne nalazi u listi, ili dok p ne postane jednak Clan-u kojeg želimo obrisati. Nakon izlaska iz petlje, ako je p jednak NULL pokazivaču izlazimo iz funkcije; inače provjeravamo postoji li prethodnik. Ako ne postoji, znači da se briše prvi član liste, što i učinimo, odnosno ako postoji, izbacije se unutarnji član što opet učinimo.

TRAŽENJE ELEMENTA U LISTI

```
ClanListe *NadiPrezimeUListi( ClanListe *Lista, char *Prezime ) {
  ClanListe *p;

  p = Lista;

  while( p != NULL ) {
    if( !strcmp(p->Prezime, Prezime) ) break;
    p = p->Slijedeci;
  }
  return p;
}
```

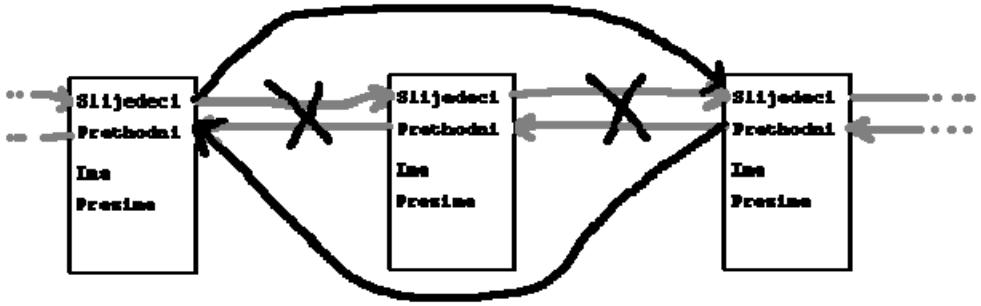
Funkcija je jako slična funkcijama za prolaz kroz listu, jer to zapravo i je baza ove funkcije. Prolaze se svi elementi liste s time da prolaz prestaje ako se pronađe član čije prezime odgovara traženom; u tom slučaju p pokazuje na taj član. Ako se član ne pronađe, petlja se odvrti do kraja i p postane NULL. Po izlasku iz petlje vrača se p.

DVOSTRUKO POVEZANE LISTE

Osnovni gradbeni element je struktura koja sadrži pokazivač na slijedeći (kao kod jednostruko povezane), ali i pokazivač na prethodni član. Kada bismo željeli naše ime i prezime pamtiti u dvostruko povezanoj listi, evo kako bismo definirali strukturu:

```
typedef struct ClanListeSt2 {
  struct ClanListeSt2 *Slijedeci;
  struct ClanListeSt2 *Prethodni;
  char Ime[20];
  char Prezime[30];
} ClanListe2;
```

Sve operacije koje smo smo prikazali za jednostruko povezane liste postoje naravno i za dvostruko povezane liste. Kao primjer ću navesti samo funkciju za brisanje.



Brisanje iz dvostruko povezane liste

```
void ObrisiIzListe2( ClanListe2 **Lista, ClanListe2 *Clan ) {
 if( Clan->Prethodni == NULL ) {
   *Lista = Clan->Slijedeci;
   (*Lista)->Prethodni = NULL;
 } else {
   Clan->Prethodni->Slijedeci = Clan->Slijedeci;
   Clan->Slijedeci->Prethodni = Clan->Prethodni;
 }
}
```

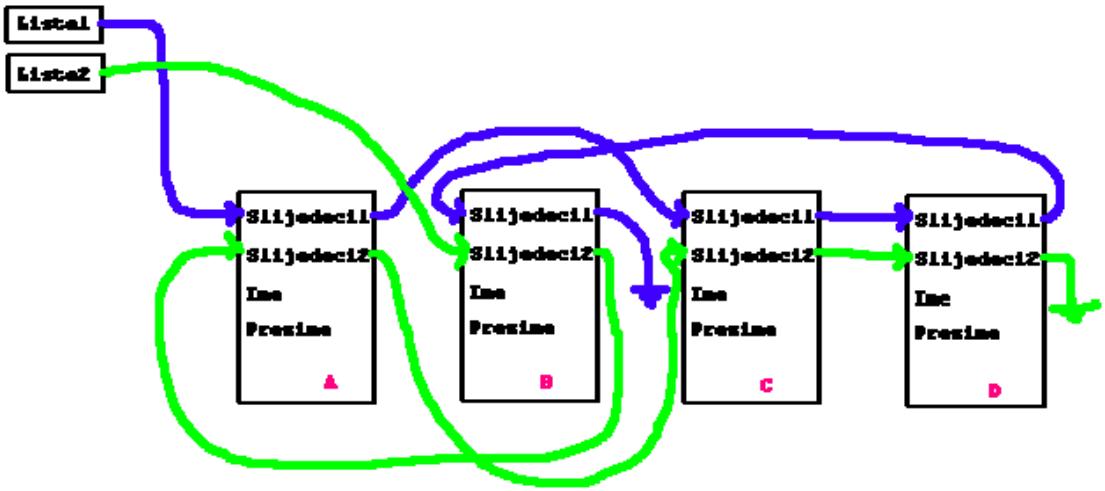
Kao što vidite, puno jednostavnije, i puno brže nego kod jednostruko povezane liste. Tamo smo morali putovati kroz listu i tražiti koji je element prethodnih elementu kojeg izbacujemo. Ovdje to ne moramo! Evo samo malog pojašnjenja rada funkcije. Provjeravamo postoji li prethodnik članu kojeg brišemo. Ako ne postoji, znači da je član prvi. Tada jednostavno postavimo pokazivač liste na Clan->Slijedeci, te sada novom prvom članu postavimo prethodnika na NULL i gotovo. Ako element nije prvi, onda imamo dvije prepravke. Prethodni član člana koji brišemo više ne pokazuje na naš član nego mora pokazivati na član iza njega, odnosno član iza onoga koji brišemo više neće pokazivati na brisani element nego mora na onaj ispred njega! Možda vas zbunjuje `Clan->Prethodni->Slijedeci` no nedajte se smesti. Clan->Prethodni je također struktura ClanListe2 i kao takva posjeduje članove Slijedeci i Prethodni. Zato je upravo napisano ispravno. Clan->Prethodni->Slijedeci je dakle pokazivac Slijedeci strukture Clan->Prethodni. Ma kužite šta hoću reći!

LISTE SA VIŠE KLJUČEVA

Uzmimo za primjer da želimo pamtiti imena i prezimena, ali na taj način da ih imamo sortirano i po imenu, i po prezimenu. Jedna mogučnost je napraviti dvije liste i sve elemente duplicirati. Jednu listu bismo tada sortirali po imenima, drugu po prezimenima. No ovakvo rješenje je previše rastrošno. Ima jedan bolji način. Definirajmo samo dva pokazivača na jednu te istu listu uz malu modifikaciju strukture samog elementa. Lista neka bude, jednostavnosti radi, jednostruko povezana.

```
typedef struct ClanListeSt3 {
  struct ClanListeSt3 *Slijedeci1;
  struct ClanListeSt3 *Slijedeci2;
  char Ime[20];
  char Prezime[30];
} ClanListe3;
```

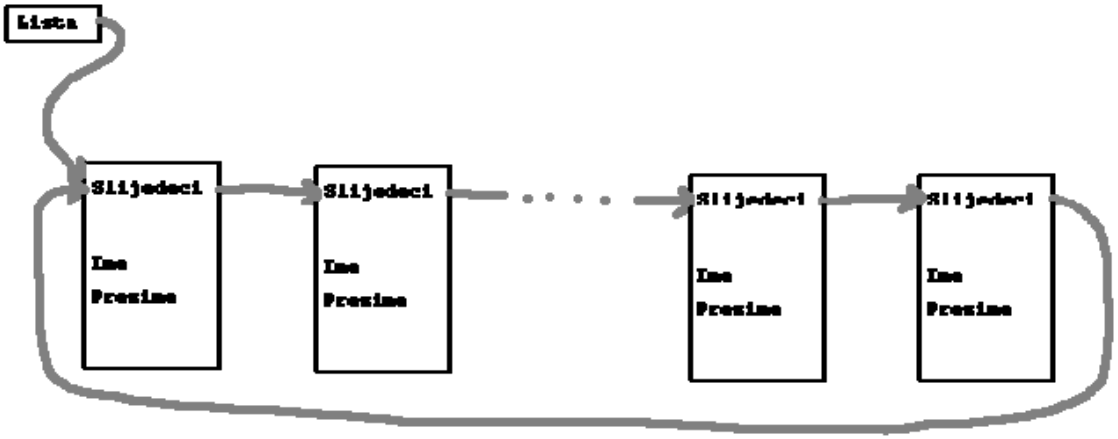
Sada ćemo za dodavanje u listu sortiranu po imenu kao Slijedeci koristiti npr. Slijedeci1, a za dodavanje u listu sortiranu po prezimenu koristiti ćemo Slijedeći2. Ovime smo duplicirali samo elemente potrebne za održavanje liste, ali nismo duplicirali i ostale podatke! Ovakva lista zove se lista sa dva ključa. Treba napomenuti da ako prolazimo listu po prvom ključu, ili je prolazimo po drugom ključu, proći ćemo sve i to iste elemente samo drugim redoslijedom. Za oslobađanje ovakve liste potrebno je brisanje provesti samo jednom, a ne za svaki ključ posebno jer se u oba prolaza kreče proz iste elemente! Dakle što smo dobili?



Lista sa dva ključa

Isto ovako mogu se kreirati i dvostruko povezane liste sa više ključeva. Pitanje je samo imate li vremena za takvo što.

I za kraj mi je ostalo objasniti što je to cirkularna lista. Pegledajte sliku ispod!



Cirkularna lista

Dakle, cirkularna lista nema kraja! Dobije se tako da se Slijedeci pokazivač 'zadnjeg' elementa liste vrati na prvi element liste. Pokušate li sada proči kroz listu običnom while(p!=NULL) petljom vrtiti ćete se do beskonačnosti! Jer ova lista nema kraja.

JEDNOSTRUKO POVEZANA LISTA SA VIŠE KLJUČEVA REALIZIRANA U DATOTECI

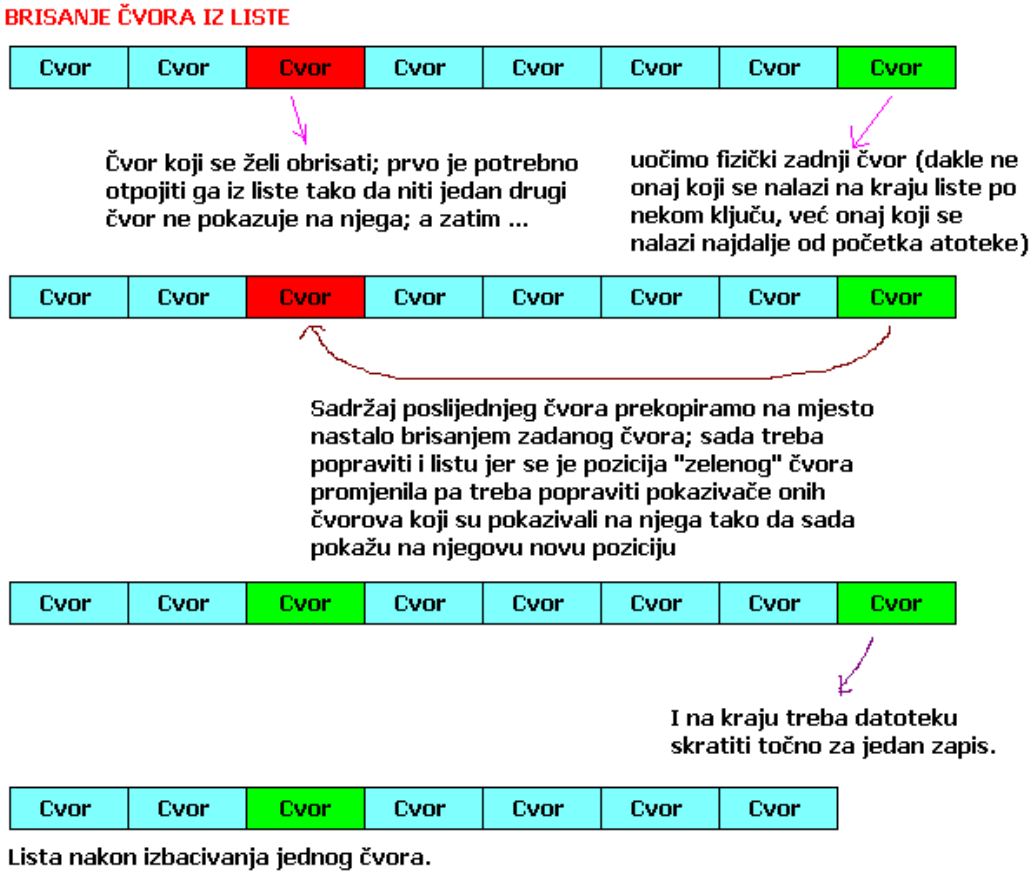
Realizacija bilo kakvih struktura u datoteci često zadaje glavobolje onima koji to moraju realizirati. Razloga je mnogo. Em nema malloc/free funkcija, nema niti normalnih pokazivača, dereferenciranje pokazivača ne postoji, ništa ne radi, sve je sporo ... No razloga za paniku jednostavno nema. Pogotovo ne u školskim primjerima. Krenimo redom.

Malloc/Free

Ovaj par funkcija često se koristi prilikom rada sa dinamičkim strukturama realiziranim u memoriji; prilikom ostvarenja istoga pri radu sa diskom, ove funkcije postaju ono zbog čega su se razvijali cijeli datotečni sustavi - noćna mora. Naime, funkcije kao takve ne postoje. Funkcije se daju ostvariti umetanjem posrednika između programa i datoteke; nekoga tko bi znao što je u datoteci slobodno, a što ne. No izrada ovakvih dijelova programa postala bi ubrzo vrlo kompleksna. Uzmimo za primjer listu pohranjenu u datoteci. Na početku dodajemo čvorove i to uvijek na kraj. Nakon toga poželimo obrisati neki čvor. Prvi korak je da ga izbacimo iz liste na taj način da ostvarimo da onaj čvor koji pokazuje na njega, pokaže na prvi čvor koji bi došao iza njega. Time smo čvor izbacili iz liste. No što je sa mjestom koje taj čvor zauzima u datoteci? Postoje bar dvije interesantne metode za riješavanje ovog problema.

1. funkcija za brisanje će potražiti koji se čvor nalazi na samom kraju datoteke, pa će njega izvaditi iz liste, fizički prekopirati na mjesto koje smo dobili eliminiranjem našeg čvora, te ponovno čvor povezati u listu, a datoteku skratiti za duljinu izbačenog čvora
2. funkcija za brisanje ne radi ništa sa nastalom rupom; funkcija za dodavanje novog čvora prolazi kroz cijelu listu i pamti gdje se koji čvor fizički nalazi u datoteci; tada nakon što prođe sve čvorove provjeri je li pronađena koja rupa u datoteci. Ako je, na to mjesto smješta novi čvor. Ako nije, novi se čvor dodaje na kraj.

Prva ideja pobliže je opisana slijedećom slikom:



Na ovaj način se osiguravamo da datoteka neće postati fragmentirana beskorisnim rupama. No kao što vidite, potrebno je raditi dosta posla da bi se pronašli čvorovi koji pokazuju na čvor koji brišemo, na čvor koji selimo i sl. pa ovaj postupak troši puno vremena.

Druga ideja vodi nas opet na trošenje vremena (a sada i memorije) jer moramo proći cijelu listu da bismo otkrili ima li gdje koja rupa. Problem je što unaprijed ne znamo niti koliko lista ima čvorova pa ne možemo unaprijed zauzeti polje zastavica kojim bismo pamtili koje su lokacije popunjene već bismo ovo morali dinamički širiti (ili možda raditi novu listu u memoriji koja bi pamtila koja su mjesta zauzeta, a bila bi sortirana po fizičkoj lokaciji mjesta).

Mi se ovim problemom nećemo zamarati jer na predmetu algoritmi i strukture podataka čovjek ne nauči dovoljno detalja da bi se upusti u izradu prvog algoritma (npr. chsize funkciju). Jednostavno ćemo ostavljati rupe, a dodavanje ćemo raditi uvijek na kraju.

Dereferenciranje

Dereferenciranje je postupak uzimanja vrijednosti kada je poznata adresa varijable. To je ono što se u C-u dobije zvijezdicom:

```
cv = first;
while( *cv != NULL ) ...
```

Pri radu sa memorijom o dereferenciranju uopće ne razmišljamo, jer po C radi sam. No pri radu sa datotekama, o dereferenciranju treba itekako voditi računa jer sve moramo raditi sami! Naime, ako znamo da je adresa varijable cv jednaka l, tada se dereferenciranje izvodi na slijedeći način:

```
fseek( f, l, SEEK_SET );
fread( &cv, sizeof(cv), 1, f );
```

Imamo i pozicioniranje (fseek), i čitanje (fread)! Zbog toga ćemo morati vrlo dobro razmisliti kada ćemo raditi dereferenciranje i koliko puta! Izmjena varijable na koju imamo pokazivača u C-u je također vrlo jednostavna (i automatska):

```
(*cv).Next = novi; // ili cv->Next = novi; ista stvar!
```

No što ovaj kod doista radi? Adresi pribraja pomak elementa Next od početka strukture na koju pokazuje cv te od te memorijske lokacije upisuje zadanu vrijednost (sadržaj varijable *novi* u našem slučaju). Kako ćemo ovo izvesti u datoteci? Neka je l adresa čvora u datoteci, a struktura čvora neka je slijedeća:

```
struct cvor {
 int Godine;
 int Visina;
 long Next;
} *cv;
```

Kod koji će izvesti gornju C-ovu naredbu glasi:

```
fseek( f, l+sizeof(cv->Godine)+sizeof(cv->Visina), SEEK_SET );
fwrite( &novi, sizeof(novi), 1, f );
```

Dakle i tu imamo pozicioniranje, i zapisivanje!

Pokazivači

Prilikom rada sa memorijom, C dopušta vrlo jednostavnu definiciju pokazivača:

struct cvor *cv;

deklarira pokazivač na jednu varijablu negdje u memoriji koja je po tipu struktura cvor. No da bismo vidjeli kako ćemo dobiti istu stvar u datoteci, idemo malo razjasniti što je to zapravo pokazivač. Pokazivač je, vjerovali Vi to meni ili ne, običan najobičniji BROJ! Ako ima vrijednost 0, tada pokazuje na nulti bajt memorije, ako ima vrijednost 1, tada pokazuje na prvi bajt memorije, i dalje mislim da ste shvatili (*za onu nekolicinu sveznalica, ovo što sam upravo rekao je točno na većini današnjih procesora, a i za x86 procesore u nekim modovima rada; ponegdje ovo baš i nije točno jer se adresa ne gleda kao jedan broj već kao dva dijela: segmentni dio i pomak, no za naša razmatranja ovakva egzotična riješenja možemo zanemariti*). Dakle, pokazivač je broj. To ćemo i mi iskoristiti i kada će nam god trebati pokazivač u datoteci, definirati ćemo ga kao long varijablu.

Radim s diskom...

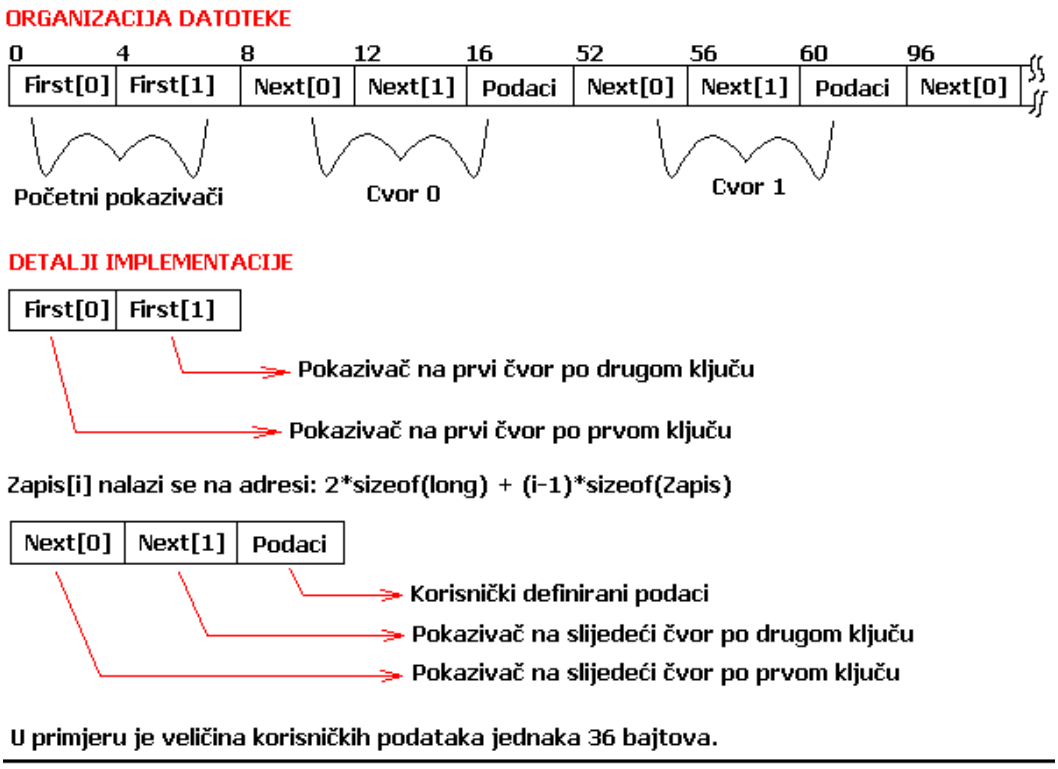
Moram razjasniti još nekoliko detalja koje ljudi znaju pomješati, prije nego što krenemo dalje. Iako se svi podaci pamte u datoteci, mi sa njima ne možemo raditi u datoteci! Procesor je sklop koji izvodi naše programe, a on zna pristupati samo memoriji! Zbog toga ćemo uvijek strukturu sa diska čitati u memoriju, tamo raditi nekakve usporedbe, izmjene i slično (i rezultat obavezno pohraniti natrag na disk!). Evo u ćemu je problem. Treba sa adrese X iz datoteke pročitati jednu strukturu i u njoj napraviti izmjenu. Riješenje koje sam nekoliko puta čuo glasi:

"Pročitamo strukturu cv u memoriju i izmjenimo ono što trebamo." Pauza.
"I?"-pitam ja. Odgovor na pitanje je zbunjen pogled.
"Što i?"

Ono što ljudi zaborave je da činjenica da su napravili izmjenu u memoriji ne znači ništa tako dugo dok tako izmjenjenu strukturu ponovno ne pohrane na disk na istu poziciju sa koje su je pročitali! Dakle nastavak ovog kratkog razgovora treba biti: *"i pozicioniram se opet na istu poziciju sa koje sam pročitao strukturu i pohranim strukturu natrag na disk."*

Jednostruko povezana lista sa dva ključa

Primjer lista sa više ključeva pokazati ćemo na primjeru liste sa dva ključa. Idemo redom. Kako ćemo organizirati datoteku da podrži izradu liste sa dva ključa?



Slika 1.

Na početak datoteke zapisati ćemo "glave" za svaki ključ. Dakle, koji je čvor prvi po prvom ključu (First[0]), te koji je čvor prvi po drugom ključu (First[1]). Da je bilo više ključeva, ovdje bismo morali dodati više ovih glava. Nakon toga pohranjujemo čvor za čvorom. Uočite da je svaki datotečni pokazivač velik 4 bajta (tj. jedan long). Struktura zapisa opet mora slijediti koncept liste sa dva ključa: imamo Next[0] što je adresa slijedećeg čvora po prvom ključu, i imamo Next[1] što je adresa slijedećeg čvora po drugom ključu. Da je bilo više ključeva, i ovdje bi moralo slijediti još Next elemenata, i to za svaki ključ po jedan! Na kraju strukture čvora nalaze se i korisnički definirani podaci; u našem primjeru to su PrIme (Prezime i ime), te datum rođenja po komponentama (godina, mjesec, dan).

```
/* Za potrebe ovog primjera definiramo strukturu cvora sa dva kljuca, i
dodatnim podacima:
   PrIme .... Prezime i ime osobe
   GodRod ... Godina rodenja osobe
   MjRod .... Mjesec rodenja (1-12)
   DanRod ... Dan rodenja (1-31)
*/
typedef struct {
 long Next[2];
 char PrIme[30];
 int GodRod;
 int MjRod;
 int DanRod;
} DDLCvor;
```

Strukturu nazivamo DDLCvor (dvoključna disk lista - cvor). Trebamo jednu funkciju koja će znati otvoriti datoteku u kojoj se nalazi lista, ili pak stvoriti novu ako ista već ne postoji (ukoliko se stvara nova, treba voditi racuna o tome da treba inicijalizirati i First[i] elemente).

```
/*
   FILE *OpenList( char *FileName, int Mode );

   Otvara datoteku u kojoj je pohranjena lista.
   Ovisno o parametru Mode datoteka se otvara:
   0 ....... za citanje i izmjene; ukoliko datoteka ne
             postoji, javlja se greska
   1 ....... za citanje i pisanje; sadrzaj datoteke se
             prilikom otvaranja brise!
   inace ... za citanje i izmjene; ukoliko datoteka ne
             postoji, biti ce kreirana; ukoliko postoji,
             sadrzaj ce biti sacuvan

*/
FILE *OpenList( char *FileName, int Mode ) {
 FILE *f;
 long l;

 if( Mode == 0 ) {
  f = fopen( FileName, "rb+" );
  if( f == NULL ) return NULL;
 }
 else if( Mode == 1 ) {
  f = fopen( FileName, "wb+" );
  if( f == NULL ) return NULL;
  l = 0;                          /* adresa je NULL */
  fwrite( &l, sizeof(l), 1, f ); /* Upisi First[0] */
  fwrite( &l, sizeof(l), 1, f ); /* Upisi First[1] */
  fseek( f, 0L, SEEK_SET );
 }
 else {
  f = fopen( FileName, "rb+" );
  if( f == NULL ) {
   f = fopen( FileName, "wb+" );
```

```
   if( f == NULL ) return NULL;
   l = 0;                          /* adresa je NULL */
   fwrite( &l, sizeof(l), 1, f ); /* Upisi First[0] */
   fwrite( &l, sizeof(l), 1, f ); /* Upisi First[1] */
   fseek( f, 0L, SEEK_SET );
  }
 }
 return f;
}
```

Trebamo i dvije funkcije koje će znati usporediti dva zapisa. Jedna će uspoređivati zapise po prezimenu i imenu, druga po datumu rođenja. Funkcije će vraćati vrijednosti manje od 0 ukoliko je prvi zapis manji od drugog, 0 ukoliko su zapisi isti, te vrijednost veću od nule ukoliko je prvi zapis veći od drugoga. Evo funkcija:

```
/* Napisimo funkciju koja ce znati usporediti dva zapisa po */
/* Prezimenu i imenu osobe...                               */
int UsporediPrIme(DDLCvor *ListElem, DDLCvor *DodElem) {
 return strcmp( ListElem->PrIme, DodElem->PrIme );
}

/* Napisimo funkciju koja ce znati usporediti dva zapisa po */
/* datumu rodjenja, i to tako da relacija veci od znaci     */
/* biti roden kasnijeg datuma.                              */
int UsporediRodjenje(DDLCvor *ListElem, DDLCvor *DodElem) {
 if(ListElem->GodRod > DodElem->GodRod ) return 1;
 if(ListElem->GodRod < DodElem->GodRod ) return -1;

 if(ListElem->MjRod > DodElem->MjRod ) return 1;
 if(ListElem->MjRod < DodElem->MjRod ) return -1;

 if(ListElem->DanRod > DodElem->DanRod ) return 1;
 if(ListElem->DanRod < DodElem->DanRod ) return -1;

 return 0;
}
```

Da ne bismo kasnije duplicirali kod, na nekim mjestima trebati će nam i pokazivač na funkciju ovog oblika. Pa evo kako se on definira:

```
int broj;                /* Broj              */
int *broj_pokazivac;     /* Pokazivač na broj */

int Usporedi_Fja(DDLCvor *ListElem, DDLCvor *DodElem);                /*
Funkcija              */
int (*Usporedi_Fja_Pokazivac)(DDLCvor *ListElem, DDLCvor *DodElem);   /*
Pokazivač na funkciju */
```

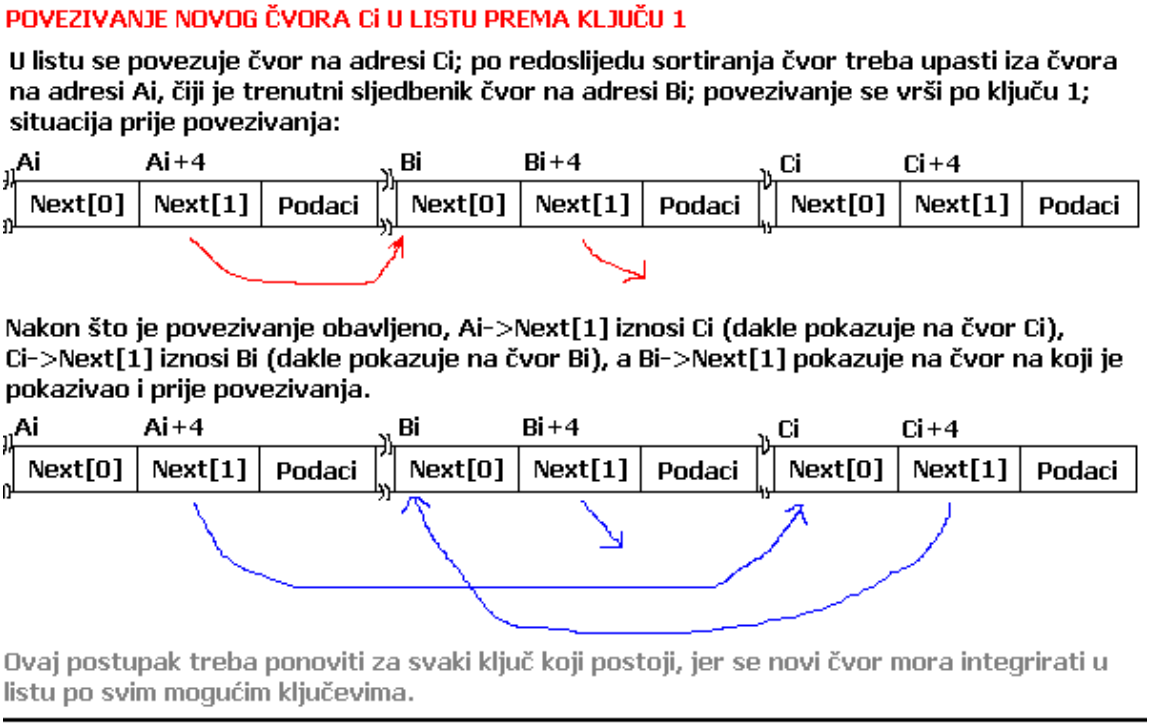
Sve što je potrebno učiniti da bi se iz prototipa funkcije dobio pokazivač na takvu funkciju jest izmjeniti ime, i staviti ga u zagrade uz jednu zvjezdicu! A kako se predaje adresa funkcije? Analogno bi pitanje glasilo kako se predaje adresa polja? Ako je int ZP[10]; deklaracija polja, tada je njegova adresa ZP! Ako je void clrscr(); deklaracija funkcije, tada je njezina adresa clrscr!

Dodavanje novog čvora u listu

Postupak se provodi u tri koraka:

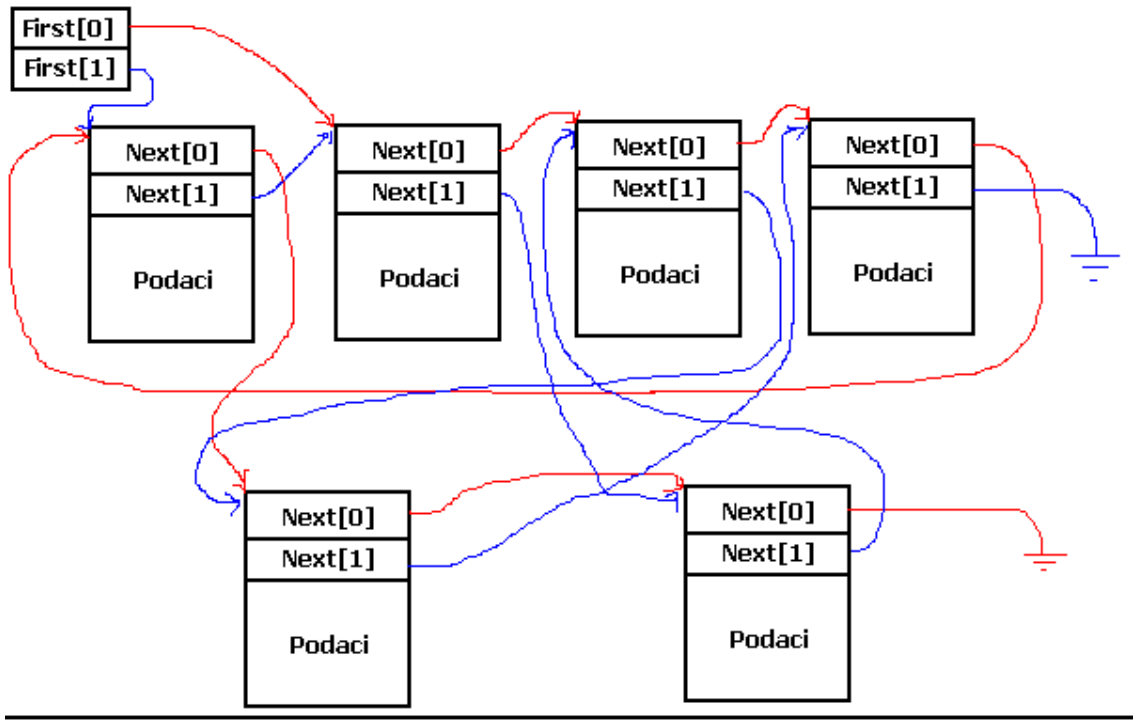
1. KORAK: Sam čvor zapiše se na kraj datoteke
2. KORAK: Izvrši se povezivanje po svim ključevima
3. KORAK: Ponovno se zapiše čvor u datoteku na istu poziciju jer je povezivanje radilo neke izmjene na čvoru

Pogledajmo kako treba izgledati postupak povezivanja u listu po ključu i:



Slika 2.

Čvor koji dodajemo zapisali smo na kraj datoteke i on je time dobio adresu Ci. Prije toga su u listi postojali već čvorovi na adresama Ai i Bi koji su bili povezani. Prema zadanom sortiranju, čvor na adresi Ci treba doći u listi iza čvora na adresi Ai a ispred čvora na adresi Bi. Na slici su opisani svi koraci koje je potrebno poduzeti. No ovaj postupak treba ponoviti za svaki ključ koji postoji! Naime, evo kako izgleda lista sa dva ključa:



Slika 3. Jednostruko povezana lista sa dva ključa; primjer.

Crvene veze pokazuju slijed po prvom ključu, plave po drugom. Povezati listu samo po jednom ključu nije ispravno jer se zadani čvor mora naći po svim ključevima. Zato i povezivanje treba obaviti za svaki ključ!

Kako izgleda funkcija koja obavlja ovo povezivanje? Evo koda:

```
/*
   int PoveziUListu(
            FILE *f,
            DDLCvor *cv,
            long cv_poz,
            int Index,
            int (*Usporedi)(DDLCvor *ListElem, DDLCvor *DodElem)
   );

   Funkcija povezuje zadani cvor prema kljucu ciji je indeks Index.
   Za sortiranje se koristi funkcija cija je adresa dana clanom Usporedi.
   Cvor koji se povezuje vec je fizicki smjesten u datoteku na poziciji
   cv_poz, a cv je pokazivac na njegovu kopiju u memoriji.
*/
int PoveziUListu( FILE *f, DDLCvor *cv, long cv_poz, int Index, int
(*Usporedi)(DDLCvor *ListElem, DDLCvor *DodElem)  ) {
 long l,t;
 DDLCvor c;

 t = Index * sizeof(long);
 fseek( f, t, SEEK_SET );
 fread( &l, sizeof(l), 1, f );
 while( l != 0L ) {
  fseek( f, l, SEEK_SET );
  fread( &c, sizeof(c), 1, f );
  if( Usporedi(&c, cv) > 0 ) break;
  t = l+Index * sizeof(long); l = c.Next[Index];
 }
 cv->Next[Index] = l;
 fseek( f, t, SEEK_SET );
 fwrite( &cv_poz, sizeof( cv_poz ), 1, f );
 fflush(f);
 return 0;
}
```

Ovo je prva funkcija koja prima pokazivač na funkciju kao parametar. Uočite kako se jednostavno poziva funkcija! Baš kao što biste pozvali i strcmp da se radi o usporedbi stringova! Evo objašnjenja funkcije. Povezujemo po ključu broj Index. Tada se varijabla First[Index] nalazi na adresi Index*sizeof(long) u datoteci. To pamtimo u varijabli t, te vršimo dereferenciranje, i dobivamo kamo pokazuje pokazivač na adresi t: to spremamo u varijablu l. I u nastavku funkcije zadržano je ovo početno značenje varijabli: t je adresa pokazivača, l je vrijednost pokazivača. Vrtimo se u while petlji tako dugo dok pokazivač ne bude NULL, ili dok ne izletimo iz petlje nekim break-om. U petlji pak ponavljamo slijedeće: l je adresa čvora, pa vršimo dereferenciranje: pozicioniramo se na adresu l, i čitamo čvor. Radimo usporedbu pročitanog čvora sa čvorom koji dodajemo, i ako je taj čvor veči od čvora koji dodajemo, izlazimo iz petlje. Inače računamo gdje koja je adresa pokazivača Next[Index] i pamtimo to u t, a l pamti vrijednost tog pokazivača, tj. pamti koja je adresa slijedećeg čvora. Jednom kada napustimo petlju, znamo da varijabla l sadrži adresu prvog većeg čvora od našeg čvora cv koji dodajemo. Zato postavljamo cv->Next[Index]=l; Uočite da je ovo napravljeno samo u memoriji! U datoteci ove izmjene još nema! To je razlog zašto će čvor cv trebati još jednom zapisati nakon što obavimo sva dodavanja. Budući da smo sada rekli da je prvi veći čvor zapravo slijedeći od cv, tada treba popraviti i pokazivač koji je pokazivao na prvi veći čvor tako da sada pokaže na adresu čvora cv u datoteci. No taj se pokazivač (l) nalazi na adresi t, pa se pozicioniramo na adresu t, i zapisujemo poziciju našeg čvora, te tjeramo operacijski sustav da ove izmjene napravi odmah (fflush).

Sada moramo još napisati funkciju koja će ostvariti prethodna tri koraka:

```
/*
   int DodajUListu( FILE *f, DDLCvor *cv );

   Funkcija DodajUListu zapisuje novi cvor na kraj liste,
   te poziva funkciju PoveziULIstu da bi novi cvor povezala
   u strukturu liste; funkciju PoveziULIstu treba pozvati onoliko
   puta koliko ima kljuceva!
*/
int DodajUListu( FILE *f, DDLCvor *cv ) {
 long l;

 /* KORAK 1a. Pozicioniraj se na kraj datoteke      */
 fseek( f, 0L, SEEK_END );
 /* KORAK 1b. Zapamti gdje si                       */
 l = ftell( f );
 /* KORAK 1c. Zapisi cvor u datoteku                */
 fwrite( cv, sizeof( DDLCvor ), 1, f );
 /* KORAK 2a. Povezi u listu po prvom kljucu        */
 PoveziULIstu( f, cv, l, 0, UsporediPrIme );
 /* KORAK 2b. Povezi u listu po drugom kljucu       */
 PoveziULIstu( f, cv, l, 1, UsporediRodjenje );
 /* KORAK 3a. Pozicioniraj se na zapamcenu poziciju */
 fseek( f, l, SEEK_SET );
 /* KORAK 3b. Zapisi ponovno izmijenjeni cvor       */
 fwrite( cv, sizeof( DDLCvor ), 1, f );
 /* KORAK 3c. I naredi trenutni zapis toga na disk. */
 fflush(f);
 return 0;
}
```

Obilazak liste po zadanom ključu

Jednom kada stvorimo listu, moramo moći nekako obići sve čvorove liste. Evo funkcije. Obilazak se vrši prema zadanom ključu.

```
/*
   void IspisiSvePoKljucu( FILE *f, int Index );

   Ispis svih zapisa sortiranih po zadanom kljucu ciji se
   indeks predaje preko parametra Index.
*/
void IspisiSvePoKljucu( FILE *f, int Index ) {
 long l,t;
 DDLCvor c;

 t = Index * sizeof(long);
 fseek( f, t, SEEK_SET );
 fread( &l, sizeof(l), 1, f );
 while( l != 0L ) {
  fseek( f, l, SEEK_SET );
  fread( &c, sizeof(c), 1, f );
  printf( "PrIme: %-30s    DatRod: %02d.%02d.%04d\n", c.PrIme, c.DanRod,
c.MjRod, c.GodRod );
  t = l+Index * sizeof(long); l = c.Next[Index];
 }
}
```

Tijelo funkcije gotovo je identično funkciji PoveziULIstu pa neću kopirati već rečeno. Dovoljno je povući slijedeću paralelu: funkcija za povezivanje je zapravo funkcija za obilazak liste koja obilazak prekida kada pronađe zapis koji traži! Time je sve rečeno.

Brisanje čvora iz liste

I postupak brisanja da se razbiti na manje zadatke.

1. Otpoji čvor iz liste po svakom ključu
2. Učini nešto sa nastalom rupom

O problemima koji se tu javljaju raspravljali smo u uvodu, kada smo objasnili kako se može izvesti korak 2 upravo navedenom algoritma. I tamo smo došli do zaključka da korak 2 nećemo implementirati. Tada nam ostaje još pokazati kako izvesti korak 1.

```
/*
   int OtpojiIzListe( FILE *f, int Index, DDLCvor *cv, long CvorPoz );

   Iz liste otpaja cvor koji se nalazi na poziciji CvorPoz; vrsi se
   otpajanje po kljucu ciji indeks sadrzi parametar Index.
*/
int OtpojiIzListe( FILE *f, int Index, DDLCvor *cv, long CvorPoz ) {
 long l,t;
 DDLCvor c;

 t = Index * sizeof(long);
 fseek( f, t, SEEK_SET );
 fread( &l, sizeof(l), 1, f );
 while( l != 0L ) {
  if( l == CvorPoz ) {
   fseek( f, t, SEEK_SET );
   fwrite( &cv->Next[Index], sizeof(cv->Next[Index]), 1, f );
   return 0;
  }
  fseek( f, l, SEEK_SET );
  fread( &c, sizeof(c), 1, f );
  t = l+Index * sizeof(long); l = c.Next[Index];
 }
 return -1;
}
```

Funkcija obilazi listu po zadanom indeksu sve dok ne pronađe čvor koji pokazuje na naš. Onog trena kada ga nađe, promjeni mu pokazivač tako da više ne pokazuje na naš, već na onaj iza našeg. I budući da i ovaj postupak treba izvesti za svaki ključ, evo funkcije koja i to čini:

```
/*
   void ObrisiIzListe( FILE *f, DDLCvor *cv, long CvorPoz );

   Funkcija otpaja zadani cvor iz svih kljuceva! cv pokazuje na cvor
   koji se zeli otpojiti, CvorPoz je njegova adresa u datoteci.
*/
void ObrisiIzListe( FILE *f, DDLCvor *cv, long CvorPoz ) {
 OtpojiIzListe( f, 0, cv, CvorPoz );
 OtpojiIzListe( f, 1, cv, CvorPoz );
 fflush(f);
}
```

Traženje čvora u listi

Traženje čvora u listi svodi se na obilazak liste koji traje tako dugo dok ili ne dođemo do kraja liste, ili ne prekinemo obilazak jer smo pronašli čvor. Problem je jedino po kojem ključu raditi obilazak? Teoretski gledamo, možemo po bilo kojem jer svaki ključ povezuje sve čvorove. No jedan ključ obično ima prednost: ako tražimo po prezimenu i imenu osobe, bilo bi zgodno obilaziti po ključu koji listu uređuje upravo po prezimenu i imenu! Zašto? Kraće je vrijeme traženja. Naime, ako dođemo do čvora koji je

"veći" od našega, ima li ga smisla dalje tražiti? Naravno da nema jer ako je u listi, onda mora biti ispred čvora koji je "veći" od njega. Kako smo mi već obilaskom utvrdili da on nije ispred, tada ga niti nema, pa traženje možemo prekinuti! Evo funkcije:

```
/*
   int PronadiCvor_PrIme( FILE *f, DDLCvor *cv, long *CvorPoz, char *PrIme );

   U listi trazi zadano prezime i ime; koristi se kljucem 0 jer on drzi
   listu sortiranu po prezimenima!

   Funkcija vraca:

   0 .... cvor je pronaden, kopija je pospremljna u varijablu na koju
          pokazuje parametar cv; pozicija cvora posprema se u CvorPoz.
   -1 ... Cvor nije pronaden!
*/
int PronadiCvor_PrIme( FILE *f, DDLCvor *cv, long *CvorPoz, char *PrIme ) {
 long l,t;
 DDLCvor c;
 int Index = 0;
 int r;

 t = Index * sizeof(long);
 fseek( f, t, SEEK_SET );
 fread( &l, sizeof(l), 1, f );
 while( l != 0L ) {
  fseek( f, l, SEEK_SET );
  fread( &c, sizeof(c), 1, f );
  if( !(r=strcmp( c.PrIme, PrIme )) ) {
   *CvorPoz = l;
   *cv = c;
   return 0;
  } else if( r > 0 ) break;
  t = l+Index * sizeof(long); l = c.Next[Index];
 }
 return -1;
}
```

Ova funkcija radi traženje isključivo po prezimenu i imenu. Dala bi se napisati i univerzalna funkcija koja bi tražila po bilo kojem podatku te bi primala pokazivač na funkciju za usporedbu, no da ne bih previše zamaglio osnovne ideje rada sa disk-listama, na temelju ove funkcije lako se napiše i bilo koja druga za pretraživanje po čemu god želite. A što se univerzalnih funkcija tiče, imate i tekst o tome, pa tko voli, neka izvoli!

Izmjena čvora liste

Svaki čvor u listu je ubačen prema nekom zadanom kriteriju. Ukoliko nakon dodavanja u listu jednostavno izmjenimo podatkovni dio nekog čvora, mogli bismo narušiti ovaj poredak, a time i spriječiti ispravan rad onih algoritama koji se oslanjaju na sortiranost liste. Zbog toga je potrebno nakon što se izvrši izmjena čvora, dotični čvor otpojiti iz liste, i ponovno ga spojiti. Naime, funkcije za spajanje pobrinuti će se da čvor sjedne na pravo mjesto u listi. Evo funkcije koja radi "Obnovu" čvora:

```
/*
   void ObnoviCvor( FILE *f, DDLCvor *cv, long CvorPoz );

   Funkcija otpaja zadani cvor iz svih kljuceva, i zatim ga ponovno
   spaja; na ovaj nacin se osigurava da cvor nakon izmjene svojih
   podataka opet bude ispravno povezan u listi.
*/
void ObnoviCvor( FILE *f, DDLCvor *cv, long CvorPoz ) {
 OtpojiIzListe( f, 0, cv, CvorPoz );
 OtpojiIzListe( f, 1, cv, CvorPoz );
```

```
 PoveziUListu( f, cv, CvorPoz, 0, UsporediPrIme );
 PoveziUListu( f, cv, CvorPoz, 1, UsporediRodjenje );
 fseek( f, CvorPoz, SEEK_SET );
 fwrite( cv, sizeof(DDLCvor), 1, f );
 fflush(f);
}
```

Otpajanje i povezivanje vrši se po svim ključevima. Kod nas ključa su dva pa imamo i dva poziva svake funkcije. Možda se pitate je li baš trebalo vršiti otpajanje i spajanje po svim ključevima? Npr. ako izmjenimo samo prezime i ime, bilo bi dovoljno popraviti listu po ključu koji listu sortira po tom podatku. No što ako je "korisnik" izmjenio više podataka? Ili što ako ključ ne sortira po jednom podatku, mego po više njih (složeni ključevi)? Generalno govoreći, treba obaviti sve! No ako ste Vi autor programa i znate što radite, naravno da možete za svaku akciju napisati ObnoviCvorX koja će popraviti samo ono što treba, jer tada Vi znate što ste radili i što treba popraviti.

BINARNA STABLA - OSNOVNE OPERACIJE

Dodavanje u binarno stablo

Definicija čvora stabla

```
typedef struct scvor {
  struct scvor *lijevi;
  struct scvor *desni;
  char Ime[50];
  int Starost;
} cvor;
```

Rekurzivan postupak

```
/*
Vrača
-1 ako čvor već postoji;
0 ako dodavanje nije uspjelo
1 ako je dodavanje uspjelo
*/

int DodajRek( cvor **glava, char *Ime, int Starost ) {
 int kamo;
 cvor *tmp;

 if( *glava == NULL ) {
  tmp = (cvor*)malloc(sizeof(cvor));
  if( tmp == NULL ) return 0;
  tmp->lijevi = tmp->desni = NULL;
  strcpy(tmp->Ime,Ime);
  tmp->Starost = Starost;
  *Glava = tmp;
  return 1;
 } else {
  kamo = strcmp(Ime, (*glava)->Ime );
  if( !kamo ) return -1;
  if( kamo < 0 )
   return DodajRek( &((*glava)->lijevi), Ime, Starost );
  return DodajRek( &((*glava)->desni), Ime, Starost );
 }
}
```

Nerekurzivan postupak:

```
/*
Vrača
-1 ako čvor već postoji;
0 ako dodavanje nije uspjelo
1 ako je dodavanje uspjelo
*/

int Dodaj( cvor **glava, char *Ime, int Starost ) {
  cvor *tmp;
  int kamo;

  while( *glava != NULL ) {
   kamo = strcmp(Ime, (*glava)->Ime );
   if( !kamo ) return -1;
   if( kamo < 0 ) glava = &((*glava)->lijevi);
   else glava = &((*glava)->desni);
  }
  tmp = (cvor*)malloc(sizeof(cvor));
  if( tmp == NULL ) return 0;
  tmp->lijevi = tmp->desni = NULL;
  strcpy(tmp->Ime,Ime);
  tmp->Starost = Starost;
  *glava = tmp;
  return 1;
}
```

Traženje u binarnom stablu

Definicija čvora stabla

```
typedef struct scvor {
 struct scvor *lijevi;
 struct scvor *desni;
 char Ime[50];
 int Starost;
} cvor;
```

Rekurzivan postupak

```
/*
Vrača pokazivač na čvor koji sadrži tražene podatke. Ako čvor ne postoji,
vrača NULL.
*/

cvor *TraziRek( cvor **glava, char *Ime ) {
  int kamo;

  if( *glava == NULL ) {
   return NULL;
  } else {
   kamo = strcmp(Ime, (*glava)->Ime );
   if( !kamo ) return *glava;
   if( kamo < 0 )
   return TraziRek( &((*glava)->lijevi), Ime );
   return TraziRek( &((*glava)->desni), Ime );
  }
}
```

Nerekurzivan postupak:

```
/*
Vraća pokazivač na čvor koji sadrži tražene podatke. Ako čvor ne postoji,
vraća NULL.
*/

cvor *Trazi( cvor **glava, char *Ime ) {
  int kamo;
  while( *glava != NULL ) {
   kamo = strcmp(Ime, (*glava)->Ime );
   if( !kamo ) return *glava;
   if( kamo < 0 ) glava = &((*glava)->lijevi);
   else glava = &((*glava)->desni);
  }
  return NULL;
}
```

Ispis stabla

Inorder - sortirani ispis stabla

```
void IspisInorder( cvor *glava ) {
 if( glava == NULL ) return;
 IspisInorder(glava->lijevi);
 printf("%s\n", glava->Ime);
 IspisInorder(glava->desni);
}
```

Preorder

```
void IspisPrerder( cvor *glava ) {
 if( glava == NULL ) return;
 printf("%s\n", glava->Ime);
 IspisPreorder(glava->lijevi);
 IspisPreorder(glava->desni);
}
```

Postorder

```
void IspisPostorder( cvor *glava ) {
 if( glava == NULL ) return;
 IspisPostorder(glava->lijevi);
 IspisPostorder(glava->desni);
 printf("%s\n", glava->Ime);
}
```

Broj članova stabla

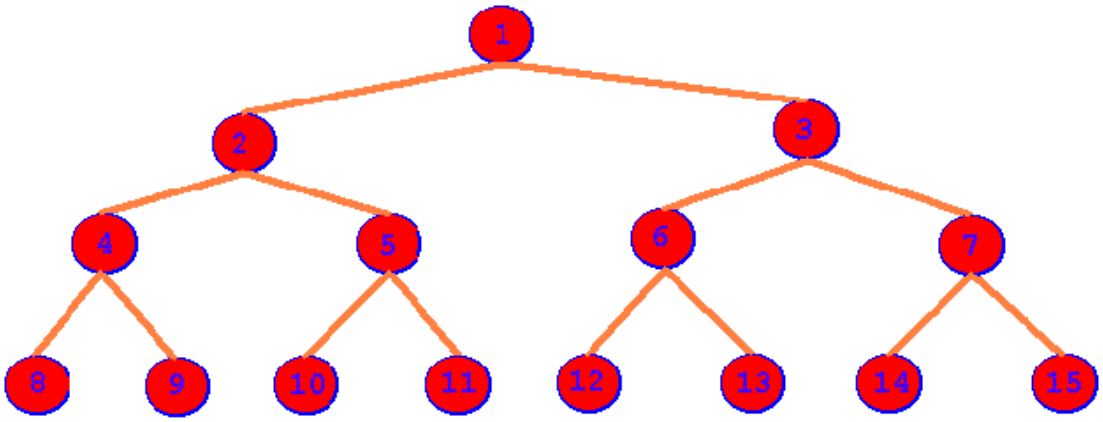
Rekurzivni postupak je najprikladniji za ove svrhe. Ideja kojom je pisan je slijedeća: stablo ima onoliko elemenata koliko ih ima lijevo, plus koliko ih ima desno, plus 1 za trenutni čvor.

N = Nlijevo + Ndesno + 1

```
int BrojClanova( cvor *glava ) {
 if( glava == NULL ) return 0;
 return BrojClanova( glava->lijevi ) + BrojClanova( glava->desni ) + 1;
}
```

POTPUNO BINARNO STABLO

Kada se binarno stablo realizira preko polja novi član jednostavno se doda na kraj polja te se zatim primjeni operacija uređaja tako da stablo i nakon dodavanja ostane onakvo kakvo želimo. Primjer ovoga radi se na predavanjima iz Algoritama i struktura podataka kada se obrađuje struktura zvana gomila. No postavlja se pitanje kako saznati kamo član treba ubaciti ako se stablo radi kao dinamička struktura. Ili još jednostavnije pitanje: dam vam pokazivač na glavu binarnog stabla, i želim da mi ispišete n-ti član tog stabla, uz pretpostavku da je stablo potpuno. Pri tome n odgovara broju člana stabla koje je numerirano kao na donjoj slici.



Potpuno binarno stablo

Osnovni problem je zapravo kako odlučiti treba li na pojedinom članu ići lijevo ili treba ići desno. Tako do 11-toga člana krećemo desno, pa lijevo, pa lijevo. Do 14-tog člana ići ćemo desno, desno, lijevo, itd.

MOGUĆE RIJEŠENJE: BINARNI BROJEVNI SUSTAV

Razmišljajući o ovom problemu dosjetio sam se da slične probleme (ili zakonitosti, kako želite) muče i binarni brojevni sustav. Naime, poznato vam je da tamo postoji sam 0 i 1. Lijevo i desno. 0 i 1.... Ma sigurno ima neke logike. I moram se pohvaliti da sam logiku i pronašao. Evo o čemu se radi. Recimo da želimo pronaći stazu do člana broj 11 potpunog binarnog stabla. U tu svrhu potrebno je broj 11 prvo pretvoriti u binarni broj. Postupak možete vidjeti na donjoj sličici. Skraćemo bismo mogli pisati i ovako: 11 = 8 + 2 + 1 tj. binarno %1011 (oznaku % koristiti ću u nastavku za označavanje binarnih brojeva). Iz ovog broja se još ne vidi kako do traženog čvora. Ono što je potrebno jest najprije odbaciti vodeću znamenku. Odbacivanjem od broja %1011 dobijemo broj %011. I odmah jedno upozorenje. Ne odbacujte vodeće nule! Trebati će nam. Nadalje prihvatimo jedan dogovor. Ako je znamenka 0, kretati ćemo se lijevo, ako je znamenka 1 kretati ćemo se desno. Pa da vidimo. Rekli smo da želimo doći do jedanaestoga člana stabla. Jedanaest smo pretvorili u binarno %1011 i odbacili smo vodeću znamenku te nam je ostalo %011. Krenimo sada od glave stabla i primjenjujmo pravilo na preostale znamenke, od prve (sa lijeve strane) do zadnje. Čitamo redom: 0 pa se od 1 krećemo lijevo na 2, zatim 1 pa se krećemo desno na 5, i konačno 1 pa se krećemo desno na 11. Isti primjer za kretanje do 14-stog člana imate na donjoj slici.

```
14:2-7, 0 ostatka
 7:2-3, 1 ostatka
 3:2-1, 1 ostatka
 1:2-0, 1 ostatka
-------------------
binarni broj 1110
bez prve znamenke 110
```

```
11:2-5, 1 ostatka
 5:2-2, 1 ostatka
 2:2-1, 0 ostatka
 1:2-0, 1 ostatka
-------------------
binarni broj 1011
bez prve znamenke 011
```

```
0 - lijevo
1 - desno
Kretanje po potpunom binarnom stablu
```

Općenito pravilo za pronalaženje vektora smjera glasi:

1. Željeni broj člana n pretvoriti u binarni broj
2. Odbaciti vodeću znamenku
3. Uvažiti kriterij kretanja:
   - 0 znači pomak u lijevo
   - 1 znači pomak u desno

Drugi uvjet je ekvivalentan sa zahtjevom da pri pretvorbi broja u binarni stanemo kada dođemo do koraka 1:2=0.

No koliko nam mjesta treba da bismo zapamtili sve binarne znamenke broja n? Broj n u binarnom zapisu imati će znamenaka $\log_2(n)+1$, to je jednako kao i $\ln(n)/\ln(2)+1$. Dakle jedna od mogučnosti je ovako dinamički izračunati koliko će znamenaka imati binarni broj, dinamički alocirati toliko bajtova memorije, pretvoriti broj, pronači član i osloboditi memoriju. No sada jedno čisto praktično pitanje: Što mislite koliko će znamenaka imati npr. milijunti član u binarnom zapisu? 1000000=%11110100001001000000, odnosno 20 znamenaka. A koliko članova stabla možemo adresirati sa 32 znamenke? Pa 2^{32} što je 4 294 967 296 članova. Znači preko 4 milijarde članova! E da mi je vidjeti računalo u kojem čete napraviti takvo stablo! Što sa ovime zapravo htio pokazati je: NE KORISTITE dinamičku alokaciju memorije. Zašto? Jer je:

- sporo
- nepotrebno
- nesigurno

Umjesto toga deklarirajte običnu varijablu, npr. char Vektor[32]; i koristite ju. Sigurno nećete tražiti 5 milijarditi element polja tako da će 32 bajta sasvim dostajati za sve vaše potrebe. I još nešto da Vas malo zainteresiram. Kompletan posao oko pretvorbe broja u binarni zapis je nepotreban (pa računala su binarna!) i ograničimo li se na 4.3 milijarde članova binarnog stabla, umjesto polja od 32 bajta dovoljna nam je jedna varijabla tipa long! Kako to? Ma neću vam baš sve otkriti! (Ili možda hoću, malo poslije...)

Source kod kako ovo realizirati slijedi:

```
typedef struct XX {
 struct XX *lijevi;
 struct XX *desni;
} BStablo;

BStablo *DajCvor( BStablo **Stablo, unsigned int n ) {
  char *v;          // Pointer na vektor kretanja
  int broj_razina;  // Broj razina potpunog binarnog stabla -1
  int pos, i;
  BStablo *t;       // Pomocni pointer za kretanje po stablu bez rekurzije

  // Ako je n <= 0, taj cvor ne postoji!
  if( n < 1 ) {
    return NULL;     // I van! Gotovo!
  }

  // Inace, ako se trazi prvi element, njega znamo!
  if( n == 1 ) return *Stablo;

  // Izracunaj broj razina stabla - 1; -1 jer nas prva
  // (ili nulta, sto got zelite) ne zanima
  // broj_razina-1= log po bazi 2 od broja clanova, ili
  // log po e od broja clanova kroz log po e od 2
  broj_razina = (int)(log( (double)n ) / log( 2. ));
```

```
  // Alociraj toliko mjesta
  v = (char*)malloc(broj_razina);

  // Popunjavanje polja krece od pocetka; dakle od pozicije 0
  pos = 0;

  // U v cemo zapravo pohraniti binarni zapis od broja n
  while( n > 1 ) {
    v[pos++] = n % 2;
    n = n / 2;
  }

  // t je prvi element stabla
  t = *Stablo;

  // krecemo od zadnjeg elementa polja jer se pri pretvorbi broja u
  // binarni zapis znamenke dobiju naopacke!!
  i = pos-1;

  // udimo u petlju, sve do prvog clana
  for( ; i >= 0; i-- ) {
   // Ako je znamenka 1, idemo desno
   if( v[i] ) {
     puts(" idemo desno! ");
     t = t->desni;            // dakle, idemo desno
   } else {
     puts( " idemo lijevo! ");
     t = t->lijevi;           // inace idemo lijevo
   }
  }

  // Sada u t imamo trazeni cvor!

  // Oslobodi polje
  free(v);

  // Vrati cvor
  return t;
}
```

Primjer je pisan na najteži mogući način: uz dinamičku alokaciju memorije; rađen je ovako da vidite kako se može napraviti na teži način. Lakši bi način bio da uz ograničenje od 4.3 milijarde članova polja(što baš i nije neko ograničenje) char *v; zamjenimo sa char v[32]; i izbacimo alokaciju i oslobađanje memorije. U nastavku je također prikazana funkcija DodajUPotpuno koja dodaje novi član u stablo ali gradeći pri tome potpuno binarno stablo. Razlika između gornje funkcije i ove je samo u tome što gornja funkcija prati vektor smjera do kraja (do zadnje znamenke). Ova funkcija pak zna da zadnji član ne postoji, pa vektor prati samo do predzadnje znamenke, a zadnju koristi da bi znala treba li novi član dodati lijevo ili desno.

```
// n je velicina postojeceg stabla! Zato se u proceduri odmah uvecava za jedan
// da bi dalo poziciju na koju moramo smjestiti cvor!
BStablo *DodajUPotpuno( BStablo **Stablo, BStablo *Cvor, unsigned int
*velicina ) {
  char *v;          // Pointer na vektor kretanja
  int broj_razina;  // Broj razina potpunog binarnog stabla -1
  int pos, i, n;
  BStablo *t;       // Pomocni pointer za kretanje po stablu bez rekurzije

  (*velicina)++;    // Povecaj velicinu za jedan jer cemo Cvor sigurno dodati

  n = *velicina;    // pomocna varijabla
```

```
  // Za svaki slucaj, kako ce ovo biti krajnji cvor, stavimo ovo na NULL
  Cvor->lijevi = NULL; Cvor->desni = NULL;

  // Ako je n == 1, stablo ima samo nas cvor! Tada JE stablo nas cvor!
  if( n < 2 ) {
    *Stablo = Cvor;  // Napravimo to!
    return Cvor;     // I van! Gotovo!
  }

  // Izracunaj broj razina stabla - 1; -1 jer nas prva
  // (ili nulta, sto got zelite) ne zanima
  // broj_razina-1= log po bazi 2 od broja clanova, ili
  // log po e od broja clanova kroz log po e od 2
  broj_razina = (int)(log( (double)n ) / log( 2. ));

  // Alociraj toliko mjesta
  v = (char*)malloc(broj_razina);

  // Popunjavanje polja krece od pocetka; dakle od pozicije 0
  pos = 0;

  // U v cemo zapravo pohraniti binarni zapis od broja n
  while( n > 1 ) {
    v[pos++] = n % 2;
    n = n / 2;
  }

  // t je prvi element stabla
  t = *Stablo;

  // krecemo od zadnjeg elementa polja jer se pri pretvorbi broja u
  // binarni zapis znamenke dobiju naopacke!!
  i = pos-1;

  // udimo u petlju, sve do drugog clana, prvi ne diramo jer on jos ne
postoji!
  // njega cemo mi dodati!
  for( ; i > 0; i-- ) {
   // Ako je znamenka 1, idemo desno
   if( v[i] ) {
     puts(" idemo desno! ");
     t = t->desni;            // dakle, idemo desno
   } else {
     puts( " idemo lijevo! ");
     t = t->lijevi;           // inace idemo lijevo
   }
  }

  // Sada u t imamo roditelja kome trebmo dodati Cvor
  // i je sada nakon petlje jednak 0; sada treba provjeriti
  // treba li dijete dodati lijevo(v[i]=0) ili desno(v[i]=1)
  if( v[i] ) { t->desni = Cvor; } else { t->lijevi = Cvor; }

  // Oslobodi polje
  free(v);

  // Vrati prvi clan stabla, tj. samo stablo
  return *Stablo;
}
```

A sada, njezino veličanstvo, ALU (ili za one koji još ne znaju, aritmetičko logička jedinica)!

Jedno retoričko pitanje: ima li potrebe binarni broj pretvarati u binarni? Koliko god vam se ovo pitanje čini čudnim, razmislite! Našoj funkciji kažete da želite n-ti član stabla. I što mi radimo? Pretvaramo taj n u binarni broj! A kako se n pamti u memoriji računala? Pa kao binarni broj! Znači da ga uopće nema potrebe pretvarati u binarni zapis kada je on već zapisan binarno. Sve što trebamo znati jest upisati računalo je li odgovarajući bit postavljen ili nije! Uzmimo opet naš primjer 14-stog člana. Broj 14 u binarnom formatu izgleda %1110. Zapišimo ga onako kako je u memoriji. Ako je varijabla tipa unsigned long, tada su to 4 bajta, tj 32 bita. Naš će broj dakle izgledati: %00000000 00000000 00000000 00001110. Ako znamo kako preskočiti vodeće nule, možemo tada odbaciti vodeču znamenku, i krenuti sa tumačenjem ostatka. Evo kako bi to napisali u C-u:

```
void IspisiStazu( unsigned long n ) {
  unsigned long b;

  if( !n ) return;

  b = (unsigned long)1 << 31;
  while( !(n & b) ) b = b >> 1;
  b = b >> 1;
  while( b ) {
   printf("%s\n", (n & b) ? "desno" : "lijevo");
   b = b >> 1;
  }
}
```

Funkcija IspisiStazu prima broj člana do kojega želimo doći, a ispisuje stazu kako do njega. Prvo provjerimo da li je n različit od nule. Ako nije, 0-ti član ne postoji, pa prekidamo sa izvođenjem funkcije. Zatim varijablu b inicijaliziramo na početnu vrijednost od 2147483648, ili 2^{31}. Vjerojatno se pitate što radi operator <<. To je operator koji je dobro poznat svima koji su se susreli sa assemblerom. On pomiče bitove varijable koja je lijevo od njega za broj mjesta desno od njega, te praznine dopunjava nulama. Evo primjera: %00110110 << 1 = %01101100. Uočite da pomak za 1 ulijevo odgovara zapravo množenju broja sa 2. Analogno operator >> pomiče stvar u desno. Isti primjer %00110110 >> 1 = %00011011. Dakle b = 1 << 31 je isto što i b = %10000000 00000000 00000000 00000000 = 2147483648. Izraz n & b je binarni operator AND; rezultat je binarni broj koji ima 1 tamo gdje oba imaju 1, inaće na tom mjestu ima nulu. Npr.

```
 %001100110011
 %010100100110 AND
 %000100100010 =
```

Uočite da je kod nas na početku postavljen samo 32 bit. n&b je 1 ako je i 32 bit od n isto jedan, inače je nula. Ukoliko je nula, b se pomiče za jedan udesno. Sada je u b postavljen samo 31 bit, i opet se provjerava je li i u n postavljen 31 bit. I tako se to vrti u petlji tako dugo dok ne dođemo stvarno do mjesta gdje je u n postavljen bit na 1. Tada je b&n različito od nule i iskače se iz petlje. Sada se sjetimo algoritma. Treba odbaciti prvi bit. U redu, pomičemo b još za jedan u desno! I sada ulazimo u novu petlju koja se vrti tako dugo dok je b razlicit od nule. Ispituje se znamenka na i-tom mjestu i tumači se rezultat. Zatim se b pomiče opet za jedan u desno sve dok b ne postane nula! Zašto b postane nula? Evo primjera %100 >> 1 = %010 >> 1 = %001 >> 1 = %000 = 0. Ovime ću i zaključiti ovo poglavlje.

UREĐENO BINARNO STABLO U DATOTECI

PODSJETNIK NA STABLA U MEMORIJI

Osnovne operacije sa binarnim stablima već smo upoznali. Vidjeli smo kako se stvara binarno stablo u memoriji, kako mu se dodaju novi elementi, te kako se stablo obilazi. Najčešće je bilo riječi o rekurzivnim funkcijama koje su to obavljale, no vidjeli smo da se dodavanje u stablo može izvesti i bez rekurzivne funkcije. No kako realizirati binarno stablo u datoteci?

Da bismo odgovorili na ovo pitanje, podsjetiti ćemo se kako smo postupali sa stablom u memoriji, i pokušati povući nekoliko paralela. U skripti iz Algoritama i struktura podataka (prof. dr. sc. Damir Kalpić, doc. dr. sc. Vedran Mornar) na strani 79-80 dana je jedna moguća izvedba funkcije za dodavanje, koju ću prepisati uz male prepravke:

```
typedef struct scv_mem {
 struct scv_mem *lijevi;
 struct scv_mem *desni;
 int x;
} cv_mem;

cv_mem *upis( cvor *glava, int x ) {
 int smjer;

 if(glava==NULL){
  glava = (cv_mem*)malloc(sizeof(cv_mem));
  glava->x = x;
  glava->lijevi = glava->desni = NULL;
 } else if( (smjer = x - glava->x) <= 0 ) {
  glava->lijevi = upis( glava->lijevi ,x );
 } else {
  glava->desni = upis( glava->desni ,x );
 }
 return glava;
}
```

Pa što kažete na ovaj algoritam? Malen, više manje kompaktan ... I očajano napisan! Pogledajte što se događa ako je glava različita od NULL: poziva se upis lijevo ili desno i ono što se vrati, ponovno se dodjeljuje natrag u glava->lijevi ili glava->desni. Ovo je naravno u redu ako je dotični pokazivač bio NULL pa sada postaje pokazivač na novi čvor. No nije u redu ako je dotični pokazivač već različit od NULL jer se time opet dodjeljuje sam sebi! Tada po povratku iz rekurzije piše zapravo glava->lijevi=glava->lijevi ili pak glava->desni=glava->desni što je potpuno nepotrebno, i troši procesorsko vrijeme. Možda vam se sada ovo čini kao cjepidlačenje jer kada radite sa memorijom, ovo se odvija relativno brzo - no zapamtite: mi želimo ovo isto raditi sa datotekom, a tada je ovo jednostavno neprihvatljivo! Pogledajmo kako se ovaj isti algoritam može poboljšati.

```
void upis( cv_mem **glava, int x ) {
 cv_mem *t;
 int smjer;

 if( *glava == NULL ) {
  t = (cv_mem*)malloc(sizeof(cv_mem));
  t->x = x;
  t->lijevi = t->desni = NULL;
  *glava = t;
  return;
 } else {
  if( (smjer = x - (*glava)->x ) <= 0 ) {
   upis(&(*glava)->lijevi,x);
  } else {
   upis(&(*glava)->desni,x);
  }
 }
}
```

U čemu je razlika? Umjesto da predajemo glavu kao pokazivač na cv_mem, mi predajemo njezinu adresu! Na ovaj način, budući da znamo njezinu adresu, kada ustanovimo da glava pokazuje na NULL, alociramo novi član i prepravljamo glavu tako da pokazuje na njega; inače se ne vrše nikakva dodjeljivanja! Zbog toga je ovaj algoritam daleko prikladniji za realizaciju binarnog stabla u datoteci.

Pogledajmo još funkciju za obilazak stabla inorder načinom.

```
void inorder( cv_mem *glava ) {
 if( glava == NULL ) return;
 inorder( glava->lijevi );
 printf("%d\n", glava->x);
 inorder( glava->desni );
}
```

Da bi koristili ovu funkciju, još nam nedostaje dio koji obično pišemo u main funkciji:

```
void main( void ) {
 cv_mem *stablo;
 int x;

 /* Inicijaliziraj stablo na NULL - prazno stablo */
 stablo = NULL;

 /* Dodaj neku vrijednost u stablo ... */
 x = 10;
 upis( &stablo, x );
}
```

MEMORIJSKA STABLA - ZAKLJUČAK

Što smo sve koristili za rad sa stablom? Imali smo:

1. glavu stabla (koju smo na početku inicijalizirali na NULL)
2. funkciju kojom smo upisivali nove elemente stabla
3. funkciju za obilazak stabla

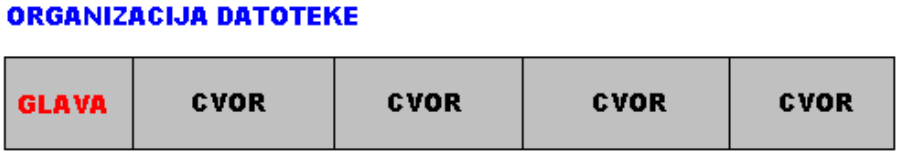
Ovo se je naravno sve nalazilo u memoriji.

STABLA U DATOTECI

Da bismo ostvarili stablo u datoteci, primjeniti ćemo potpuno iste algoritme, ali sa jednom razlikom: algoritmi će biti prilagođeni tako da umjesto u memoriju po podatke posežu u datoteku. Što ćemo sve koristiti za rad sa ovakvim stablom?

1. glavu stabla (koju ćemo na početku inicijalizirali na NULL)
2. funkciju kojom ćemo upisivali nove elemente stabla
3. funkciju za obilazak stabla

Evo kako ćemo organizirati našu datoteku:



Na samom početku datoteke nalaziti će se pokazivač na prvi čvor stabla, kojega ćemo na početku programa inicijalizirati na NULL. No sada je već i došlo vrijeme da kažem nešto o tim "pokazivačima". Naše stablo se ne nalazi u memoriji već u datoteci. Pitanje koje se postavlja je kako adresirati podatke u datoteci? Najjednostavniji način za ovo daju nam funkcije fseek / ftell. Kako one adresiraju datoteku? Odgovor glasi: udaljenost od početka datoteke. Tako ćemo i mi koristiti istu metodu. Za dodavanje novog čvora stabla pozicionirati ćemo se na na kraj datoteke, pitati ftell gdje se nalazimo i na to mjesto upisati novi čvor, a rezultat ftell funkcije smatrati ćemo adresom čvora.

UPIS NOVOG PODATKA U BINARNO STABLO

U nastavku ću pokazati kako se može upisati podatak u stablo u datoteci. Primjenjen će biti algoritam koji smo dobili prilagodbom algoritma za memorijska stabla. I odmah jedna napomena. Ako radimo sa memorijom, i funkciji proslijedimo pokazivač na adresu prvog čvora:

```
void upisi( cv_mem **glava, ... )
```

tada nam je sasvim jasno što se događa ako napišemo npr:

```
if( *glava == NULL ) radi nešto ...
```

Pa što se događa? Budući da zna gdje se nalazi traženi podatak (glava je njegova adresa), procesor obavlja čitanje traženog podatka sa te adrese i uspoređuje to sa NULL pokazivačem. Dakle, procesor je za nas obavio jedno čitanje! Pri radu sa datotekom ovo nažalost neće biti moguće, već ćemo to čitanje morati obaviti mi eksplicitno! Našu funckiju ćemo definirati ovako:

```
void dodaj_bsf( long a_glava, ... )
```

pri čemu će a_glava biti, dakako, adresa u datoteci gdje se nalazi pokazivač glava! A taj ćemo pokazivač onda pročitati tako da se pozicioniramo na adresu a_glava, i pozovemo funkciju fread da nam pročita glavu:

```
long glava;
fseek(f,a_glava,SEEK_SET);
fread(&glava,sizeof(glava),1,f);
```

I sada ćemo ispitati if(glava == NULL) radi nešto. Da ne duljim, evo kako se to da ostvariti:

```
typedef struct {
 long lijevi;
 long desni;
 int x;
} cvor_bsf;
void dodaj_bsf( long a_glava, FILE *f, int x ) {
 cvor_bsf t; long glava;

 fseek(f, a_glava, SEEK_SET);
 fread( &glava, sizeof(glava), 1, f );
 if( glava == 0 ) {
  t.lijevi = t.desni = 0;
  t.x = x;
  fseek( f, 0L, SEEK_END );
  glava = ftell(f);
  fwrite( &t, sizeof(t), 1, f );
  fseek( f, a_glava, SEEK_SET );
  fwrite( &glava, sizeof(glava), 1, f );
  return;
 } else {
  int smjer;

  fseek( f, glava, SEEK_SET );
  fread( &t, sizeof(t), 1, f );
  smjer = x - t.x;
  if( smjer < 0 )
    dodaj_bsf( (char*)&t.lijevi-(char*)&t + glava, f, x );
  else if( smjer > 0 )
    dodaj_bsf( (char*)&t.desni-(char*)&t + glava, f, x );
 }
}
```

Kao što možete primjetiti, pri kraju koda uočljiva je još jedna razlika sa memorijskim algoritmom. Naime, ako znamo da glava pokazuje na neki čvor, tada do elemenata tog čvora dođemo jednostavno: glava->lijevi i sl. No sa datotekama to opet nije moguće! Zbog toga smo na početku programa definirali varijablu t tipa cvor_bsf te nam ona služi da bismo se pozicionirali u datoteci na mjesto na koje pokazuje glava, i sa tog mjesta čitamo čvor u varijablu t, te tada imamo pristup do elemenata t.lijevi i sl. Nadalje, kada smo radili sa memorijom, adresu pokazivača dobili smo &(*glava)->lijevi. No kako ćemo dobiti adresu istog pokazivača u datoteci? Jednostavno! Kombinirati ćemo malo memorije i malo datoteka. Ono što znamo jest da u datoteku upisujemo čvorove, tj. strukture. U memoriji u trenutku izvođenja funkcije imamo također jednu takvu strukturu: varijablu t. E sada, varijabla t počinje na adresi &t, a t.lijevi se nalazi na adresi &t.lijevi. Možemo zaključiti da se t.lijevi nalazi adresa od t.lijevi - adresa od t bajtova od početka varijable t! Ali baš toliko bajtova daleko će se nalaziti i u datoteci od početka strukture koju smo pročitali! Naravno, još zbog C-ove adresne aritmetike obje adrese moramo ukalupiti u char* da bismo dobili razliku u bajtovima. Zbog toga svega, ako znamo da se tražena struktura u datoteci nalazi na poziciji glava, tada se njezin element lijevi nalazi na poziciji:
(char*)&t.lijevi - (char*)&t + glava
a element desni na adresi:
(char*)&t.desni - (char*)&t + glava

Još bi trebalo reći kako smo realizirali malloc funkciju! Stvar je vrlo jednostavna. Funkcijom fseek odemo na kraj datoteke, pa funkciju ftell pitamo gdje smo - to nam je trenutna adresa. Sada na tu adresu upišemo novi čvor funkcijom fwrite koja će automatski proširiti datoteku za broj upisanih bajtova! I time je rješen problem.

Ostatak algoritma identičan je onom za rad sa memorijom.

RASPRŠENO ADRESIRANJE

Kada govorimo o pretraživanju podataka, obično nam kao najbrži mogući napamet pada algoritam binarnog pretraživanja. No postoji još jedan algoritam, koji ponekad može biti i daleko brži od bilo kakvog pretraživanja, uključujući ovdje i već spomenuto binarno pretraživanje. Evo o čemu se radi. Zamislite ormar sa 100 ladica, i u svakoj ladici se nalazi jedan papir sa brojem od 1 do 100. Dalje, pretpostavimo da su papiri sortirani, tako da se u prvoj ladici nalazi onaj sa brojem jedan, u drugoj sa brojem dva, ..., a u zadnjoj onaj sa brojem 100. Pokušajmo u tom ormaru pronaći papir sa brojem 1.

1. BINARNO PRETRAŽIVANJE

Algoritam binarnog pretraživanja zahtjeva sortiranost podataka, i taj je uvjet ispunjen. Algoritam dalje kaže slijedeće: pogledaj papir u onoj ladici koja se nalazi u sredini. Pa ako su ladice numerirane od 1 do 100, to je pedeseta ladica. Otvaramo, i vidimo da je u ladici papir sa brojem 50; taj broj je veći od traženoga, pa slijedeću provjeru radimo u ladici koja se nalazi točno na pola puta od početka do upravo provjerene pedesete ladice. Gledamo dakle 25-tu ladicu. Broj je i opet prevelik, pa provjeravamo ladicu koja se nalazi opet na pola puta, tj. 12-tu ladicu, itd. sve dok ne dođemo do prve ladice gdje konačno pronalazimo traženi papir. Ne znam jeste li brojali, ali ladice smo otvarali 6 puta. Sada se vratimo u realnost, gdje su naši podaci smješteni na disku. Svakom otvaranju ladice odgovara jedno pozicioniranje glave diska, čitanje minimalno jednog sektora podataka (512 ili čak 1024 bajtova), te povrat glave u početni položaj. I tako 6 puta! Za veće količine podataka ovaj broj naravno postaje sve veći i veći (dobro, ovisnost jest logaritamska, ali svejedno, i logaritamska funkcija raste!).

2. RASPRŠENO ADRESIRANJE

Ovaj algoritam polazi od jedne činjenice, a to je da znamo relaciju koja povezuje traženi podatak i njegovu lokaciju. Vratimo se primjeru sa paririma i ormarom. Koja relacija povezuje broj papira i broj ladice u kojoj se papir nalazi? Pa to je *broj_ladice=broj_papira*. I kako se onda vrši pretraživanje podataka? Iz poznate relacije odredimo položaj traženog podatka, i jednostavno provjerimo je li on tamo, ili taj podatak uopće ne postoji! Dakle, za papir sa brojem 50 potrebno je prema relaciji otvoriti 50-tu ladicu, i provjeriti je li unutra papir ili nije. Ako jest, onda je sigurno sa brojem 50 (što provjerimo), inače ako nije, tada sigurno papir sa brojem 50 uopće nije u ormaru! Da vidimo koliko smo sada puta otvarali ladice? Točno jednom!

POVRATAK U REALNOST

Ovdje opisan postupak raspršenog adresiranja bio je čista idealizacija. U realnost su naravno moguće i ponešto kompliciranije situacije. Jedna od mogučnosti jest da imamo npr. 200 papira, a samo 100 ladica. Tada ćemo morati u svaku ladicu staviti po dva papira. Mogli bismo ustanoviti jednostavnu relaciju: *broj_ladice=zaokruži(broj_papira/2)*. Sada bismo pri traženju određenog papira morali nakon što otvorimo ladicu provjeriti oba unutra prisutna papira. No kada već znamo ladicu, puno je lakše provjeriti unutra prisutne papire, nego stalno trčati od ladice do ladice i pogađati gdje bi se podatak mogao nalaziti. Pogledajmo još malo kompliciraniju situaciju. Imamo 200 papira, ali sa brojevima od 1 do 300 (naravno da svi brojevi nisu istorišteni). Broj ladica je 100, a relacija glasi *broj_ladice=zaokruži(broj_na_papiru/3)*. Ovom će relacijom svaki papir biti poslan u jednu od 100 ladica koje su nam na raspolaganju. Naravno, kako imamo 200 papira, u svaku bi ladicu trebalo staviti po dva papira. Zaboravimo na trenutak traženje papira, već pogledajmo kako bismo uopće papire poslagali u ladice, tako da gornja relacija bude zadovoljena. Uzmimo za primjer da su neki od brojeva na papirima 100,150,21,70,20,32,22,... Pogledajmo kamo ćemo postaviti koje papire. Prvi papir je sa brojem 100, pa on ide u ladicu broj 100/3=33.3 što zaokružimo na 33. Slijedeći papir po istoj formuli ide u 50-tu ladicu, zatim slijedeći u sedmu, pa u 23., pa papir sa brojem 20 ide u sedmu (sada su u sedmoj ladici dva papira, i više nema mjesta za nove papire!), dalje papir sa brojem 32 ide u 11. ladicu, zatim broj 22 ide u sedmu ladicu... Pa baš i ne ide u sedmu jer je sedma već popunjena! Što sada učiniti? Kamo ćemo staviti ovaj papir? Odgovora na ovo pitanje ima nekoliko, a jedan od najučestalijih je pokušajmo ga smjestiti u prvu slijedeću slobodnu ladicu, iako mu tamo zapravo i nije mjesto. Ova odluka utjecati će na način na koji ćemo vršiti pretraživanje, no na to ćemo se vratiti kasnije. Znači, nakon što smo otkrili da više nema mjesta u ladici broj 7, pokušajmo u ladici broj 8. U njoj ima mjesta, pa papir stavljamo unutra. I tako redom dalje popunimo sve preostale papire. Što smo ovime dobili? Potrpali smo sve papire u ladice, ali smo pri tome narušili zadanu relaciju.

Ona više ne vrijedi točno, ali ipak daje približnu lokaciju gdje bi se traženi papir mogao nalaziti. Probajmo pronači onaj papir broj 22 koji nam je uzrokovao probleme. Po formuli, od bi se trebao nalaziti u ladici broj 7; provjeravamo: nema ga. No kako je ladica puna, pretpostavljamo da je on mogao preći u slijedeću ladicu. Otvaramo ladicu broj 8, i pronalazimo traženi papir. Koji bi zaključak bio da papir nismo pronašli niti u ladici broj 8? Ako je ladica puna, trebalo bi pogledati u ladicu broj 9. No ako ladica nije puna, dakle sadrži manji broj papira od maksimalnog, zaključak bi bio da papir ne postoji niti u jednoj ladici! Naime, ako papir nije stao u 7 ladicu, tada bismo ga pokušali staviti u 8; kako 8 ladica nije popunjena, to bismo i uspjeli! No kako papir nije u osmoj ladici, zaključak bi morao biti da papir nije prisutan!

KOMPLIKAŽA BROJ 2

A što ako (uvijek ima ako!) dopustimo brisanje podataka? Tj. ako prilikom traženja nekog papira nakon što ga nađemo, dopustimo si slobodu da ga odnesemo iz ladice? Kakve to posljedice ima na traženje? Tada stvar postaje noćna mora. Naime, ovo je najgori mogući slučaj raspršenog adresiranja. Zašto? Recimo da smo pri popunjavanju papir broj 22 htjeli staviti u ladicu broj 7, no nismo mogli jer je bila puna, te smo papir uspjeli pohraniti u ladicu broj 8. Pri tome se u ladici 7 nalaze papiri 21 i 20. Obavimo sada prvo traženje za papirom broj 20. Pronalazimo ga u ladici broj 7 i vadimo. Sada je u ladici broj 7 ostao samo papir broj 21, te ladica nije više puna. Potražimo sada papir broj 22. Gledamo u sedmoj ladici: nema ga, ali ladica nije popunjena. Sada više ne smijemo zaključiti da papir ne postoji! Jer on se nalazi u slijedećoj ladici! Dakle, ako je dopušteno vađenje papira (tj. u stvarnim aplikacijama brisanje podataka), pri traženju podataka nužno se moraju provjeriti sve raspoložive lokacije, dakle kod nas cijeli ormar!

ZAŠTO JE RASPRŠENO ADRESIRANJE DOBRO

U gornjim primjerima vidjeli smo kakvi se sve slučajevi mogu javiti pri raspršenom adresiranju:

- upis iz prve
- upis u jednu od slijedećih ladica (tzv. preljev)
- brisanje podataka

Prvi slučaj je najlakši za realizaciju, zadnji očito najteži. No svi ovi slučajevi se u prosjeku ponašaju vrlo dobro. Naime, relacija koja nam daje lokaciju gdje se podatak nalazi, uvelike nam ubrzava proces traženja, jer ne moramo pogađati gdje se podatak nalazi (kao npr. binarno pretraživanje). Zatim, podatke nitko ne mora sortirati (što je relativno spor posao), već ih je dovoljno samo poslagati po nekoj logici (što je daleko brže). I konačno, ideja da se podaci grupiraju u ladice znatno doprinosi brzini cijelog postupka, jer je rad sa diskom puno sporiji od rada sa memorijom. Zato je puno isplativije odjednom pročitati n podataka koliko ih stane u jednu ladicu, pa to u memoriji pretražiti, nego n puta čitati po jedan podatak sa diska. Postavlja se pitanje kako odabrati veličinu ladice u realnim aplikacijama. Sjetimo se kako radi disk. On pri svakom čitanju pročita barem jedan cijeli sektor, no nije mu problem niti pročitati nekoliko sektora koji su smješteni u nizu, jer se sve to događa od jednom; dakle sa samo jednim pozicioniranjem glave diska. Logičan je zaključak da za veličinu ladice odaberemo višekratnik broja sektora! Što bi se godilo ako odaberemo sektor i pol za veličinu ladice? Pri čitanju prve ladice bili bi pročitani 1 i 2 sektor, od čega bi mi dobili sektor i pol podataka. Pri čitanje druge ladice bili bi pročitani sektori 2 i 3, a mi bi opet dobili sektor i pol podataka. No vjerojatno ste uočili da se sektor dva čita dva puta (i to cijeli!). To je krajnje nepraktično i sporo! I to je razlog za savjet da se kao veličina ladice koristi cijeli višekratnik broja sektora; dakle, 1 sektor, 2 sektora, 3 sektora, itd. Inače, već sam spomenuo da od diska do diska veličina sektora varira između 512 i 1024 bajtova (512 za starije diskove, 1024 za neke novije modele). Ovdje u primjerima koristiti ću se brojkom od 512 bajtova.

OPTIMIZACIJA

U opisu raspršenog adresiranja susreli smo se sa nekim od problema, i vidjeli smo zašto se oni javljaju. Da bi raspršeno adresiraje težilo svom idealnom modelu, potrebno je osigurati:

1. što veći broj podataka po ladici, da se smanji broj pristupanja disku
2. što manji broj podataka po ladici, da se smanji vrijeme potrebno za pronalaženje podatka u ladici
3. što veći broj raspoloživih ladica

Sigurno se pitate kako udovoljiti zahtjevu 1 i 2 istovremeno. Pakosni bi sigurno rekli: deriviraj i izjednači sa nulom! Ali zapravo to i nije daleko od istine. Naime, da bi raspršeno adresiranje bilo efikasno, oba

zahtjeva su sasvim logična. Disk je spor pa treba smanjiti broj pristupanja disku; dakle, trebamo puno podataka u svakoj ladici. S druge strane, traženje podataka u velikom polju je sporo, dakle, u svakoj ladici treba imati idealno jedan podatak, realno, nekoliko njih. U praksi ćemo uzimati doista samo nekoliko podataka po ladici. Treći zahtjev je, nadam se, jasan.Ako imamo jako veliki broj raspoloživih ladica, tada će relacija koja razmješta podatke po ladicama manje podataka smještati u iste ladice, te će biti puno manje nepoželjnih preljeva. No broj ladica je naravno ograničen i odozdo. On mora biti minimalno takav sa može primiti sve podatke koje želimo upisati.

PARAMETRI RASPRŠENOG ADRESIRANJA

Kada jednom krenemo u realizaciju raspršenog adresiranja, potrebno je odabrati neke fiksne parametre. To su redom:

- BPS - broj bajtova po sektoru
- BSL - broj sektora po ladici
- BP - broj podataka sa kojima želimo raspolagati
- BPL - broj podataka po ladici
- FS - faktor sigurnosti
- UBL - ukupni broj ladica sa uračunatim faktorom sigurnosti

Uzmimo za primjer da su podaci koje pohranjujemo strukture:

```
typedef struct {
  char Ime[20];
  char Prezime[20];
} Podatak;
```

Tada će se parametri računati prema relacijama:

```
#define BPS = 512  // po dogovoru
#define BSL = 2    // odabiremo npr. 2 sektora po ladici
#define BP = 4000  // zadano nam je da trebamo 4000 ucenika pohraniti u tablicu
#define BPL = BPS * BSL / sizeof(Podatak)  // ovoliko podataka stane u jednu ladicu
#define FS = 1.3   // dodajemo 30% više mjesta od potrebnoga da smanjimo vjerojatnost
preljeva
#define UBL = (int)((double)BP / BPL * FS) // ukupan broj ladica u tablici
```

Ove relacije su univerzalne i vrijede za svaku tablicu raspršenog adresiranja. Ono što je promjenjivo, to su parametri BPS, BSL, BP, FS. Dalje, potrebno je definirati jednu ladicu koja će biti zajednička za čitav program.

```
char cLadica[BPS*BSL];               // polje koje je veliko točno kao jedna ladica
Podatak *Ladica = (Podatak*)cLadica; // pomočni pokazivač koji polje promatra kao polje Podataka
```

PRIPREMA TABLICE RASPRŠENOG ADRESIRANJA

Da bismo vršili upisivanje podataka u novu tablicu, potrebno je prvo stvoriti cijelu tablicu i sve podatke inicijalno postaviti na prazne! To će nam obaviti procedura InitTablica. Budući da ćemo podatke pretraživati po Prezimenu, trebati će se dogovoriti kada ćemo smatrati da je podatak prazan. Uzmimo da je podatak prazan ako mu je prezime nul-string; dakle prazan string. Funkcija slijedi:

```
void InitTablica( FILE *f ) {
  int i;

  // obrisi ladicu
  memset( Ladica, 0, sizeof(cLadica) );  // umjesto sizeof(cLadica) moglo je biti i BPS*BSL
                                         // samo ne sizeof(Ladica) !!! jer je Ladica pokazivač
  fseek( f, 0, SEEK_SET );               // pozicioniraj se na početak datoteke
  for( i = 0; i < UBL; i++ ) {           // brisi ladicu po ladicu
    fwrite( Ladica, sizeof(cLadica), 1, f );
  }
}
```

Nakon inicijalizacije možemo krenuti sa upisom podataka. Za to ćemo zadužiti funkciju Upisi.

```
int Upisi( FILE *f, char *Ime, char *Prezime ) {
  int poc, lokacija;
  int Nastavi, Upisano;
  int i;

  // pozovi funkciju koja na temelju Prezimena generira lokaciju kamo treba smjestiti
podatak
  lokacija = GenerirajLokaciju( Prezime );
  poc = lokacija;    // Zapamti pocetnu lokaciju

  // Jos nismo upisali podatak, pa
  Upisano = 0;

  do {
    fseek( f, lokacija * sizeof(cLadica), SEEK_SET );  // pozicioniraj se na ladicu
    fread(Ladica, sizeof(cLadica), 1, f);              // procitaj ladicu
    for( i = 0; i < BPL; i++ ) {                       // provjeri sve podatke
     if( Ladica[i].Prezime[0] == '\0' ) {              // Ako je podatak prazan
      strcpy(Ladica[i].Prezime, Prezime);              // Upisi novi podatak
      strcpy(Ladica[i].Ime, Ime );                     // ..................
      fseek( f, lokacija * sizeof(cLadica), SEEK_SET );// pozicioniraj se na ladicu
      fwrite(Ladica, sizeof(cLadica), 1, f);           // zapisi ladicu
      Upisano = 1;                                     // Upisali smo podatak!
      break;                                           // gotovi smo!
     }
    }
    if( Upisano ) break;
    lokacija = (lokacija+1) % UBL;                     // pređi na slijedecu ladicu
(vidi ***)
    if( lokacija == poc ) Nastavi = 0;
  } while( Nastavi );
  return Upisano;
}
```

(***) Ako imamo 20 ladica, tada se one numeriraju od 0 do 19. Pri upisu podatka moramo pronači mjesto gdje ćemo podatak upisati, ako nam ne stane u ladicu koju je formula dala. Tako npr. ako smo trebali upisati podatak u ladicu broj 18, a nismo uspjeli, pokušavamo u 19. Ako ni tu ne uspijemo, treba pokušati sa slijedećom, a to je ladica broj 0! Nakon toga ide ladica broj 1, ... sve dok ne dođemo do ladice sa kojom smo krenuli. Ako smo doista došli do početne ladice, tada je sve puno, i nema mjesta za novi podatak, te izlazimo iz petlje. Upisano uvećavanje ladice može se napisati u dvije linije:

```
 lokacija++;
 if( lokacija >= UBL ) lokacija = 0;
```

što se kraće da napisati uz malo matematike (i operacije modulo) kao lokacija = (lokacija+1) % UBL;

Algoritam za traženje odgovarajućeg podatka ovisi o tome želi li dopustiti brisanje podataka ili ne. Naime, ako ne dopuštamo brisje podataka, tada sa traženjem možemo stati ili kada pronađemo podatak, ili kada naiđemo na prazan podatak (vidi raspravu prije). No ako dopuštamo brisanje, tada moramo pretraživati cijelu tablicu. Funkcija slijedi:

```
int Trazi( FILE *f, char *Prezime ) {
  int poc, lokacija;
  int Nastavi, Nadeno;
  int i;

  // pozovi funkciju koja na temelju Prezimena generira lokaciju gdje bi se podatak
mogao nalaziti
  lokacija = GenerirajLokaciju( Prezime );
  poc = lokacija;    // Zapamti pocetnu lokaciju

  // Jos nismo pronasli podatak, pa
  Nadeno = 0;

  do {
    fseek( f, lokacija * sizeof(cLadica), SEEK_SET );  // pozicioniraj se na ladicu
    fread(Ladica, sizeof(cLadica), 1, f);              // procitaj ladicu
    for( i = 0; i < BPL; i++ ) {                       // provjeri sve podatke
     if( strcmp(Ladica[i].Prezime, Prezime) ) {        // Ako je podatak pronaden
       Nadeno = 1;                        // postavimo zastavicu; i je indeks podatka!
       break;                                          // izlaz
     }
     // Ovaj uvjet ispod treba ispitivati ako brisanje nije dopušteno;
     // inace treba izbaciti slijedeca 4 retka!
     if( Ladica[i].Prezime[0] == '\0' ) {              // Ako je podatak prazan
      Nadeno = -1;                                  // Tada sigurno trazeni ne postoji
      break;                                          // te smo gotovi!
     }
    }
    if( Nadeno ) break;
    lokacija = (lokacija+1) % UBL;             // pređi na slijedecu ladicu (vidi ***)
    if( lokacija == poc ) Nadeno = 0;
  } while( Nastavi );
  return (Nadeno == 1) ? i : -1;
}
```

Do sada sam već dva puta koristio funkciju GenerirajLokaciju bez da sam išta objašnjavao. Nadam se da ste shvatili da ta funkcija glumi onu relaciju između podatka i njegove ladice koju sam prehodno opisivao. Važnost ove funkcije je izuzetna, jer ako loše napišete ovu funkciju, tada će se javljati puno preljeva, tj. podaci će se slabo raspršivati. A tada vam sve prednosti raspršenog adresiranja padaju u vodu. Izgled funkcije uvelike ovisi o tipu podatka iz kojega se generira ključ (odnosno lokacija; termin ključ češće se koristi). Jedan od najjednostavnijih primjera sam već naveo: imamo 100 ladica i integer podatke od 0 do 199 koje treba razmjestiti u ladice. Jedno moguće rješenje je koristiti relaciju:
lokacija = broj / 2;
Ovo će sigurno sve podatke razbacati po ladicama od nulte (broj 0) do 99-te (broj 199).
Ako kao podatak imamo string, tada bismo ključ mogli dobiti također vrlo jednostavno. Evo primjera:

```
int GenerirajLokaciju2( char *Prezime ) {
 unsigned char i=0;
 while( *Prezime ) { i += *Prezime++; }
 return (int)((double)i / 255. * (UBL-1));
}
```

Ovdje dva prikazana algoritma relativno su loša, jer daju dosta preljeva, tj. slabo raspršenje. Danas su razvijeni algoritmi koji daju puno bolje rezultate, no o tome drugom prilikom. U skripti iz predmeta Algoritmi i strukture podataka također je navedeno nekoliko ovih algoritama, pa prolistajte.

REKURZIVNE FUNKCIJE

Početi ću sa definicijom naslova. Rekurzivna funkcija je svaka funkcija koja poziva samu sebe. Neki ljudi ovome dodaju još jedan uvjet, a to je da svaka rekurzivna funkcija mora svoj posao završiti u konačnom vremenu, tj. u konačno mnogo koraka. No osobno smatram da je ovaj dodatak višak. Naime, ukoliko tako isprogramirate kôd da završi u konačno mnogo koraka, to je u redu. Ukoliko pak isprogramirate takvu funkciju koja nikada neće završiti, pa to je vaša stvar! No ono što uzrokuje to nezavršavanje opet je su rekurzivni pozivi same funkcije. No pređemo li u realnost, svaka rekurzivna funkcija će završiti u konačno mnogo koraka. To slijedi prosto iz fizičkih ograničenja računala na kojemu se funkcija izvodi. Svaki rekurzivni poziv troši dio memorije. Stoga se u realnosti mogu dogoditi dvije stvari. Ili će funkcija na vrijeme prestati pozivati samu sebe, ili će računalo ostati bez memorije. Kako god okrenete, stvar će jednom stati! No budimo ozbiljni. Rekurzivne funkcije treba pisati tako da je svakim novim pozivom posao koji obavlja funkcija sve bliži kraju, tj. konvergira konačnom rješenju. Takve će rekurzivne funkcije sigurno završiti sa poslom u konačno mnogo koraka.

KAKO PISATI REKURZIVNE FUNKCIJE

Prvo moramo vidjeti može li se naš posao shvatiti nekako rekurzivno. Ako je odgovor potvrdan, tada ima smisla pisati rekurzivnu funkciju za izvršavanje tog posla. Inače je bolje odmah odustati. Ukoliko u našem poslu ipak otkrijemo rekurzivnost, to znači da se naš posao može razbiti u korake na taj način, da svaki korak bude sastavljen od dva dijela: (1) jednog nezavisnog i (2) jednog koji izgleda baš kao i početni posao. Tada se nezavisni dio izračuna (očito nerekurzivno, bar što se tiče početne funkcije), a za ostatak posla pozove se ponovno sama funkcija. Evo o čemu govorim. Uzmimo za primjer svima dobro poznatu funkciju za izračun faktorijele. To je relacija:

n! = n * (n-1) * (n-2) * .... * 3 * 2 * 1

No već površnim pogledom na gornju relaciju vidimo slijedeće: ako grupiramo članove na ovaj način:

n! = n * [(n-1) * (n-2) * .... * 3 * 2 * 1]

Tada je:

n! = n * (n-1)!

I to je baš forma koja nam je trebala! Nadam se da uočavate dva dijela o kojima sam govorio. Za **n!** treba izračunati **(n-1)!**, što je posao (2), te to pomnožiti sa **n** što je posao (1). Kada se ovdje prestaje sa rekurzivnim pozivima? Pa pokušajmo izračunati funkciju za par brojeva. Npr.

```
3! = 3 * 2!  ... imamo rekurzivni poziv
2! = 2 * 1!  ... imamo rekurzivni poziv
1! = 1       ... nema rekurzije!
```

Iz priloženog primjera vidimo da rekurzivni pozivi prestaju onog trena kada argument funkcije postane jednak 1, tj. matematika nam govori i da je 0! isto tako jednako 1, pa možemo reći da rekurzivni pozivi prestaju za 1 ili za 0. No što je sa negativnim brojevima? Koliko je (-1)! ? Relacija kojom smo opisali način izračuna faktorijele za negativne brojeve divergira, jer ako od -1 počnemo oduzimati jedinice, nikada nećemo stići do nekog pozitivnog broja ili nule, te će i naša rekurzivna funkcija, kada je jednom napišemo, biti divergentna za takve brojeve. Zato ćemo se dogovoriti o još jednom uvjetu. Proširimo gornju definiciju faktorijele sa uvjetom koji matematički možda i nije točan, ali će nam osigurati da nam poziv funkcije ne zamrzne računalo ako se pozove sa negativnim brojem. Recimo da ako je argument funkcije manji od 0, funkcija je jednaka 0. Isprogramirati ćemo dakle funkciju:

n'! = { n! za n>=0 tj. 0 za n<0}

Faktorijele se definiraju za cijele brojeve. No i jako brzo rastu. Stoga ćemo se odlučiti za tip funkcije koji omogučava najveći raspon cijelih pozitivnih brojeva: `unsigned long`. U praksi se često umjesto ovog tipa koristi `double`, jer iako on može prikazivati i decimalne brojeve, ipak mu je opseg daleko veći od `unsigned long` tipa. Nadalje, prisjetimo se do koje smo rekurzivne relacije došli za izračun n!. Bila je to relacija: n! = n * (n-1)!, uz dodatne uvjete: 0! = 1! = 1 te k! = 0 za k<0, što nam je osiguralo da funkcija konvergira rješenju za svaki argument. Tada ćemo rekurzivnu funkciju napisati na slijedeći način. Prvo ćemo ispitati sve uvjete. Tek ako oni nisu zadovoljeni, koristiti ćemo rekurzivne pozive.

```
unsigned long Faktorijela( int Broj ) {
  if( Broj < 0 ) return 0;  /*Ako je broj negativan, vrati nulu*/
  if( Broj < 2 ) return 1;  /*Ako je broj jednak 0 ili 1, vrati 1*/
  return Broj * Faktorijela( Broj-1); /*Inace koristi rekurzivnu relaciju*/
}
```

Samom argumentu funkcije dodjelili smo tip `int` iz čisto praktičnih razloga. Naime, najveći broj koji on može prikazati je 32767, a faktorijela tog broja je daleko veća od brojeva prikazivih unsigned long tipom. Zapravo, kada malo razmislimo, najveći broj čija faktorijela stane u unsigned long tip čiji je opseg 4294967295, jest broj 12, jer je 12! = 479001600, dok je već 13! = 6227020800 što prekoračuje mogućnost prikaza unsigned long tipa. Iz ovog razmatranja vidljivo je da smo za tip argumenta umjesto int mogli staviti i najmanji tip kojim C uopće raspolaže, a to je `char`, čiji je opseg i dalje puno prevelik u odnosu na brojeve koji se smiju pojaviti kao argumenti.

JOŠ NEKOLIKO PRIMJERA REKURZIVNIH FUNKCIJA

U nastavku ću navesti još nekoliko primjera koji se mogu riješiti rekurzivno. Neki od njih se nikada ne koriste, jer postoje daleko jednostavniji načini od ovdje navedenih a koji ne uključuju rekurziju. No proučite primjere da biste shvatili o čemu se radi. Vidjeti ćete sa su sve funkcije neovisno o problemu pisane uvjek na isti način. Prvo se ispitaju uvjeti a teko potom se poziva rekurzija ukoliko je to potrebno. Evo primjera.

1. Rekurzivno izračunati duljinu znakovnog niza.

Rješenje: Rekurzivna relacija glasi: duljina(text) = 1 + duljina(text+1), duljina(text)=0 ako je *text==0. Rječima bismo ov relaciju mogli izraziti ovako: niz je dugačak onoliko koliko ima ostatak niza plus jedan za trenutni znak. Dodatno, ako je trenutni znak jednak nul-terminatoru, tada je taj niz dugačak nula znakova.

```
int duljina( char *text ) {
  if( *text == '\0' ) return 0;
  return 1 + duljina(text+1);
}
```

2. Rekurzivno izračunati n-ti Fibonaccijev broj.

Rješenje: Rekurzivna relacija glasi: fib(n) = fib(n-1)+fib(n-2), fib(0)=0, fib(1)=1.

```
int fib( int n ) {
  if( n == 0 ) return 0;
  if( n == 1 ) return 1;
  return fib(n-1)+fib(n-2);
}
```

3. Rekurzivno izračunati n-tu potenciju broja, pri čemu je n cijeli broj.

Rješenje: Rekurzivna relacija se razlikuje za dva slučaja. Ako je potencija pozitivna, tada je $baza^{potencija} = baza*baza^{potencija-1}$. Npr. $2^3 = 2*2^2$.
Ako je potencija manja od nule, tada je $baza^{potencija} = baza^{potencija+1}/baza$. Npr. $2^{-3}=2^{-2}/2$. Dodatno, ukoliko je potencija jednaka 0, rezultat je 1.

```
double potenc( double baza, int pot ) {
  if( pot == 0 ) return 1;
  if( pot > 0 ) return baza * potenc(baza, pot-1);
  if( pot < 0 ) return pot(baza, pot+1) / baza;
}
```

4. Rekurzivno pronaći koliko se puta u nizu znakova nalazi zadani znak.

Rješenje: Zadani znak se nalazi u nizu onoliko puta koliko ga ima u ostatku niza plus jedan ako je trenutni znak jednak traženomu odnosno plus nula ako je trenutni znak različit od traženoga. Dodatno, znak se pojavljuje nula puta u praznom nizu!

```
int broji( char *text, char znak ) {
  if( *text == '\0' ) return 0;
  if( *text == znak ) return 1+broji(text+1,znak);
  else return broji(text+1,znak);
}
```

5. Rekurzivno pronaći poziciju zadnjeg pojavljivanja znaka u zadanom nizu. Vratiti adresu tog znaka, ili NULL ako znak nije pronađen.

Rješenje:

Ako iza trenutne pozicije postoji zraženi znak, onda treba vratiti njega. Inaće, ako je trenutni znak jednak traženomu, treba vratiti njegovu poziciju, inaće vratiti NULL. Dodatno, u praznom nizu nema znaka pa vraćamo NULL.

```
char *trazizad( char *text, char znak ) {
  char *t;

  if( *text == '\0' ) return NULL;
  t = trazizad(text+1,znak);
  if( t != NULL ) return t;
  if( *text == znak ) return text;
  return NULL;
}
```

6. Usporedi dva znakovna niza (a i b) rekurzivno. Vratiti: 0, a==b; >0, a>b; <0, a<b.

Rješenje:

Ako se trenutni znakovi razlikuju, vrati rezultat. Ako su pak isti, provjeri ostatak nizova. Dodatno, prekini kada jedan niz završi.

```
int usporedi( char *a, char *b ) {
  int t;

  t = (int)*a - (int)*b;
  if( t != 0 ) return t;     /*ako su razliciti, vrati rezultat*/
  if( !*a ) return 0;        /*inače su isti; provjeri jesmo li na kraju*/
  return usporedi( a+1, b+1);/*ako nismo na kraju, provjeri preostale
znakove*/
}
```

7. Nađi najveći broj u polju od n elemenata.

Rješenje:

Potrebno je usporediti trenutni broj sa najvećim iz ostatka polja. Ako je on veći, vrati njega; inaće vrati najveći iz ostatka polja. Dodatno, u polju koje ima samo jedan element, taj je ujedno i najveći.

```
int najveci( int *p, int n ) {
  int q;

  if( n == 1 ) return *p;  /*u polju od jednog elementa, očito je taj i
najveći*/
  q = najveci( p+1,n-1);   /*inace pogledaj u ostatku polja*/
  if( *p > q ) return *p;  /*ako je trenutni vec od svih u ostatku, vrati
njega*/
  return q;                /*inace vrati najveci iz ostatka*/
}
```

Opaska: ako je p pokazivač na trenutni element polja, tada je slijedeči element polja na p+1 i to polje je za jedan manje od početnog! Zato u rekurzivnom pozivu ide (p+1, n-1).

SORTOVI

U svijetu računala često se pojavljuje potreba za sortiranjem. Stoga ću u nastavku pokazati neke od osnovnih algoritama. Pri tome ćemo obratiti pažnju i na vrijeme koje je potrebno da bismo zadane elemente sortirali. Za početak ćemo se držati elemenata koji su pohranjeni slijedno, tj. u polje. Za elemente ćemo uzeti cijele brojeve (integer tip podataka). No kada se malo izvježbamo, pokazati ćemo i kako napraviti univerzalne funkcije. Pa krenimo redom.

Kako radimo sa poljima, treba nam funkcija koja nam može izgenerirati polje slučajnih brojeva. Kod funkcije je slijedeći:

```
void GenArray( int *array, int n ) {
int i;

for( i = 0; i < n; i++ )
array[i] = rand();
}
```

Što bi napravio mali Ivica kada mu kažete da napiše program koji će sortirati polje brojeva (od manjih prema većima)? Pa vjerojatno nešto poput ovoga:

1. *kreni od prva dva susjedna elementa i radi:*
   - *Zamjeni elemente ukoliko je potrebno, tako da prvi od ta dva postane manji.*
   - *Ponavljaj prethodnu točku i za sve preostale elemente (tj. za drugi i treći, treći i četvrti, ...)*
2. *Ponovi točku 1 n-1 puta (n je broj elemenata polja)*

Da ovaj algoritam radi, vrlo se lako možemo uvjeriti jednostavnim primjerom. Uzmimo polje sa 5 elemenata i pokažimo rad algoritma. Neka je polje brojeva slijedeće: { 9, 8, 7, 5, 2 }

Korak 1, prvi puta

```
Ispitivanje 1. i 2. elementa ==> radi zamjenu: 8, 9, 7, 5, 2
Ispitivanje 2. i 3. elementa ==> radi zamjenu: 8, 7, 9, 5, 2
Ispitivanje 3. i 4. elementa ==> radi zamjenu: 8, 7, 5, 9, 2
Ispitivanje 4. i 5. elementa ==> radi zamjenu: 8, 7, 5, 2, 9

Završno stanje ==> 8, 7, 5, 2, 9
```

Korak 1, drugi puta

```
Ispitivanje 1. i 2. elementa ==> radi zamjenu: 7, 8, 5, 2, 9
Ispitivanje 2. i 3. elementa ==> radi zamjenu: 7, 5, 8, 2, 9
Ispitivanje 3. i 4. elementa ==> radi zamjenu: 7, 5, 2, 8, 9
Ispitivanje 4. i 5. elementa ==> nema zamjene: 7, 5, 2, 8, 9

Završno stanje ==> 7, 5, 2, 8, 9
```

Korak 1, treći puta

```
Ispitivanje 1. i 2. elementa ==> radi zamjenu: 5, 7, 2, 8, 9
Ispitivanje 2. i 3. elementa ==> radi zamjenu: 5, 2, 7, 8, 9
Ispitivanje 3. i 4. elementa ==> nema zamjene: 5, 2, 7, 8, 9
Ispitivanje 4. i 5. elementa ==> nema zamjene: 5, 2, 7, 8, 9

Završno stanje ==> 5, 2, 7, 8, 9
```

Korak 1, četvrti puta

```
Ispitivanje 1. i 2. elementa ==> radi zamjenu: 2, 5, 7, 8, 9
Ispitivanje 2. i 3. elementa ==> nema zamjene: 2, 5, 7, 8, 9
Ispitivanje 3. i 4. elementa ==> nema zamjene: 2, 5, 7, 8, 9
Ispitivanje 4. i 5. elementa ==> nema zamjene: 2, 5, 7, 8, 9

Završno stanje ==> 2, 5, 7, 8, 9
```

Kao što vidite, uzeli smo, možemo slobodno reći, najgori slučaj: polje koje je sortirano suprotno od željenog načina. Da algoritam radi, vidi se iz završnog rezultata. No da vidimo neke detalje. Zašto je dovoljno n-1 prolaza da bi polje postalo sortirano? Pa neka se najveći broj nalazi na početku. Koliko zamjena je potrebno da bi on došao na kraj polja? ***n-1***, dakako. Pogledajmo i neke detalje vezane uz rad algoritma. Koliko smo puta usporedili dva broja da bismo vidjeli da li ih treba zamijeniti? 16 puta! Imali smo n-1=4 prolaza, i u svakom prolazu n-1=4 ispitivanja, što ukupno iznosi ***(n-1)*(n-1)=n^2 - 2*n + 1***, što je kod nas 4*4=16. Koliko smo radili stvarnih izmjena? Pa u prvom prolazu radili smo sve izmjene (n-1,4) i time je na zadnje mjesto stigao najveći broj koji se više nikada neće micati. Zbog toga smo u drugom prolazu radili jednu izmjenu manje (n-2,3), a ujedno je i na predzadnje mjesto stigao drugi po veličini broj koji se više neće micati. U trećem koraku radimo zato još jednu izmjenu manje (n-3,2), i u zadnjem koraku imamo samo jednu izmjenu (n-4,1). Kako to izgleda u općenitom prikazu preko formule? ***(n-1)+(n-2)+(n-3)+ ... + (n-(n-1)) =(n-1)+(n-2)+(n-3)+ ... + (1)=n*(n-1)/2***. Dakle broj izmjena je opet kvadratna funkcija, i to uvijek manja od broja provjera (n-1)*(n-1), osim za slučaj kada je n=2, jer tada imamo jednu provjeru i jednu izmjenu. No sa ovim podacima treba postupati oprezno, jer ovo što smo izveli, izveli smo za najgori mogući slučaj! Realno će broj provjera ostati isti, ali će zato broj stvarnih izmjena biti nešto manji, sve ovisno o danim elementima polja. Stavimo još ove podatke u tablicu:

Funkcija sortiranja: Sort0		
Broj elemenata polja	n	
Broj provjera elemenata	(n-1)*(n-1) = n^2 - 2*n + 1	kvadratna ovisnost
Broj zamjena elemenata	n*(n-1)/2	kvadratna ovisnost

Evo i funkcije koja radi ovo sortiranje:

```
void Sort0( int *array, int n ) {
 int i,j;
 int temp;

 for( i = 0; i < n-1; i++ ) {
  for( j = 0; j < n-1; j++ ) {
   if( array[j] > array[j+1] ) {
    temp = array[j];
    array[j] = array[j+1];
    array[j+1] = temp;
   }
  }
 }
}
```

Morate priznati da kod izgleda trivijalno; dvije for petlje i još četiri linije koda. Za prvu ruku, odlično! Ali... Naravno, uvijek postoji nekakav ali. U današnje doba najvažnija je brzina! Koliko će algoritmu trebati da sortira određeno polje? Pa brzina rada algoritma ovisi naravno o broju instrukcija koje mikroprocesor mora izvesti da bi obavio sortiranje, a taj je pak broj izravno proporcionalan broju ispitivanja u našem algoritmu, i broju izmjena koje treba obaviti. Kako je ovo kvadratna funkcija od broja elemenata polja, stvar je jako spora! Pitanje glasi, možemo li nešto od svega ovoga smanjiti? Pogledajmo možemo li smanjiti broj ispitivanja odnosa dva elementa. Pokušajmo malo preraditi gornji algoritam u nešto slično. U gornjem algoritmu smo vidjeli da nakon svakog prolaza na zadnja mjesta isplivaju najveći brojevi koji se poslije ne mijenjaju. Njih onda niti ne treba ispitivati! To ćemo postići tako da modificiramo graničnik unutarnje for petlje sa *n-1* na *n-1-i*, ili jednostavnije rečeno, svaki prolaz skratimo za jedno ispitivanje. Evo algoritma:

1. *kreni od prva dva susjedna elementa i radi:*
   - *Zamjeni elemente ukoliko je potrebno, tako da prvi od ta dva postane manji.*
   - *Ponavljaj prethodnu točku i za sve preostale elemente (tj. za drugi i treći, treći i četvrti, ...), osim za zadnjih k, pri čemu je k jednak broju trenutnog ponavljanja točke 1 umanjen za 1 (tj. za prvi prolaz k=0, itd.)*
2. *Ponovi točku 1 n-1 puta (n je broj elemenata polja)*

Sada se odmah vidi i slijedeće: prvo napravimo n-1 usporedba, zatim n-2 jer znamo da je zadnji broj najveći, itd. sve dok ne dođemo do jedne usporedbe! Broj usporedba dakle iznosi: n*(n-1)/2, što je ujedno jednako broju zamjena u najgorem slučaju. Modificirana funkcija je slijedeća:

```
void Sort01( int *array, int n ) {
 int i,j;
 int temp;

 for( i = 0; i < n-1; i++ ) {
  for( j = 0; j < n-1-i; j++ ) {
   if( array[j] > array[j+1] ) {
    temp = array[j];
    array[j] = array[j+1];
    array[j+1] = temp;
   }
  }
 }
}
```

Možemo pogledati i rad funkcije na polju iz prošlog primjera {9, 8, 7, 5, 2}

```
Korak 1, prvi puta

Ispitivanje 1. i 2. elementa ==> radi zamjenu: 8, 9, 7, 5, 2
Ispitivanje 2. i 3. elementa ==> radi zamjenu: 8, 7, 9, 5, 2
Ispitivanje 3. i 4. elementa ==> radi zamjenu: 8, 7, 5, 9, 2
Ispitivanje 4. i 5. elementa ==> radi zamjenu: 8, 7, 5, 2, 9

Završno stanje ==> 8, 7, 5, 2, 9
```

```
Korak 1, drugi puta

Ispitivanje 1. i 2. elementa ==> radi zamjenu: 7, 8, 5, 2, 9
Ispitivanje 2. i 3. elementa ==> radi zamjenu: 7, 5, 8, 2, 9
Ispitivanje 3. i 4. elementa ==> radi zamjenu: 7, 5, 2, 8, 9

Završno stanje ==> 7, 5, 2, 8, 9
```

```
Korak 1, treći puta

Ispitivanje 1. i 2. elementa ==> radi zamjenu: 5, 7, 2, 8, 9
Ispitivanje 2. i 3. elementa ==> radi zamjenu: 5, 2, 7, 8, 9

Završno stanje ==> 5, 2, 7, 8, 9
```

```
Korak 1, četvrti puta

Ispitivanje 1. i 2. elementa ==> radi zamjenu: 2, 5, 7, 8, 9

Završno stanje ==> 2, 5, 7, 8, 9
```

Dakle, za ovaj sort nam vrijedi:

Funkcija sortiranja: Sort01		
Broj elemenata polja	n	
Broj provjera elemenata	n*(n-1)/2	kvadratna ovisnost
Broj zamjena elemenata	n*(n-1)/2	kvadratna ovisnost

Dobro, sada smo pokazali kako možemo smanjiti broj usporedbi dva elementa polja. Možemo li kako smanjiti broj međusobnih zamjena elemenata? Idemo malo razmisliti. Ideja malog Ivice dovela nas je do jednog načina sortiranja polja. No, ima mali Ivica još ideja u rukavu. Pogledajmo čega se je još dosjetio:

1. *pronađi najmanji element u cijelom polju, i stavi ga na prvo mjesto, a element sa prvog mjesta stavi na njegovo mjesto*
2. *zatim ponavljaj korak 1 n-1 puta, s time da u pretrazi zanemariš prvih k članova (pri čemu je k jednak broju trenutnog ponavljanja točke 1 umanjen za 1)*

Implementacija algoritma je također trivijalna, a zasniva se na inverziji prethodnih algoritama. Kod prvog algoritma smo vidjeli da svakim prolazom na zadnja mjesta ispliva po jedan najveći element polja. Ovaj pak algoritam kaže ovo: nađimo svakim prolazom po jedan najmanji broj i stavimo ga u polje krenuvši od početka polja. Evo realizacije:

```
void BubbleSort1( int *array, int n ) {
 int i,j,m;
 int temp;

 for( i = 0; i < n-1; i++ ) {
  // pretpostavimo da je najmanji element baš i-ti
  m = i;
  for( j = i+1; j < n; j++ ) {
   // ako si našao manji od pretpostavljenog, zapamti mu indeks
   if( array[m] > array[j] ) m=j;
  }
  // ako je najmanji element razlicit od pretpostavljenog, zamjeni ih
  if( m != i ) {
   temp = array[i];
   array[i] = array[m];
   array[m] = temp;
  }
 }
}
```

Koliko sada imamo usporedbi elemenata? (n-1)+(n-2)+...+1 = n*(n-1)/2, dakle, ništa novoga. A koliko radimo stvarnih izmjena elemenata? **n-1**. Samo n-1 (i to u najgorem slučaju). Izgleda dosta dobro u odnosu na n*(n-1)/2. Pogledajmo rad ovog algoritma na već poznatom primjeru polja {9, 8, 7, 5, 2}

```
Korak 1, prvi puta

Početno stanje: 9, 8, 7, 5, 2
Pretpostavljeni minimum: 1-i element.

Ispitivanje 2. elementa ==> novi minimum: 2-i element.
Ispitivanje 3. elementa ==> novi minimum: 3-i element.
Ispitivanje 4. elementa ==> novi minimum: 4-i element.
Ispitivanje 5. elementa ==> novi minimum: 5-i element.

Mijenja se 1-i element sa 5-im elementom.

Završno stanje ==> 2, 8, 7, 5, 9
```

```
Korak 1, drugi puta

Početno stanje: 2, 8, 7, 5, 9
Pretpostavljeni minimum: 2-i element.

Ispitivanje 3. elementa ==> novi minimum: 3-i element.
Ispitivanje 4. elementa ==> novi minimum: 4-i element.
Ispitivanje 5. elementa ==> stari minimum: 4-i element.

Mijenja se 2-i element sa 4-im elementom.

Završno stanje ==> 2, 5, 7, 8, 9
```

```
Korak 1, treći puta

Početno stanje: 2, 5, 7, 8, 9
Pretpostavljeni minimum: 3-i element.

Ispitivanje 4. elementa ==> stari minimum: 3-i element.
Ispitivanje 5. elementa ==> stari minimum: 3-i element.

Nema izmjene.

Završno stanje ==> 2, 5, 7, 8, 9
```

```
Korak 1, četvrti puta

Početno stanje: 2, 5, 7, 8, 9
Pretpostavljeni minimum: 4-i element.

Ispitivanje 5. elementa ==> stari minimum: 4-i element.

Nema izmjene.

Završno stanje ==> 2, 5, 7, 8, 9
```

Zaključak glasi:

Funkcija sortiranja: BubbleSort1		
Broj elemenata polja	n	
Broj provjera elemenata	n*(n-1)/2	kvadratna ovisnost
Broj zamjena elemenata	n-1	linearna ovisnost

Gore dani algoritmi moguće su implementacije Bubble sort algoritma, koji predstavlja osnovnu ideju kako nešto sortirati. No osnovno je, kako se pokazuje, jako sporo. Zbog toga se traže algoritmi koji ovo mogu izvesti malo brže. Jedan od njih je i čuveni Heap sort.

HEAP (Gomila) i HEAP SORT

Heap je struktura podataka - binarno stablo - pri čemu vrijedi da za svaki čvor *K* i relaciju **R**, *K* i lijevo dijete od *K* su u relaciji **R**, i *K* i desno dijete od *K* su u relaciji **R**. Relacija **R** može biti proizvoljna, a često se koristi relacija *veće od* ili pak *manje od*.

Evo primjera sa cijelim brojevima:

Prikazan je *heap sa relacijom veće od*. Svaki roditelj veći je od svoje djece. Ovakav heap često se naziva **max heap**, jer se na vrhu nalazi najveći element. Ukoliko bi relacija bila *manje od*, dobili bismo **min heap**.

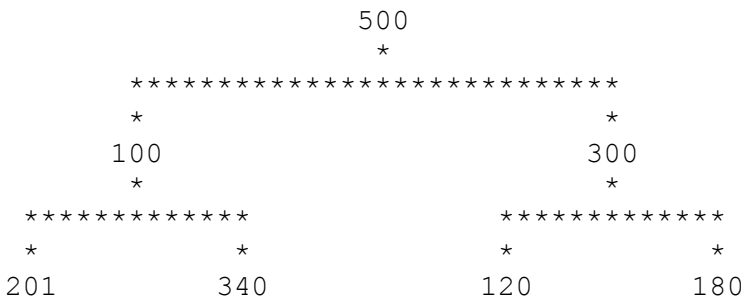
Ova će nam struktura biti korisna prilikom sortiranja, pa pogledajmo najprije kako neko zadano polje elemenata složiti u heap. Prvo, kako uopće polje elemenata promatrati kao binarno stablo? Dva su načina:

1. Ako se roditelj nalazi na indeksu n, djeca su mu na indeksima 2*n (lijevo) i 2*n+1 (desno). Prvi element pohranjen je na indeksu n=1.
2. Ako se roditelj nalazi na indeksu n, djeca su mu na indeksima 2*n+1 (lijevo) i 2*n+2 (desno). Prvi element pohranjen je na indeksu n=0.

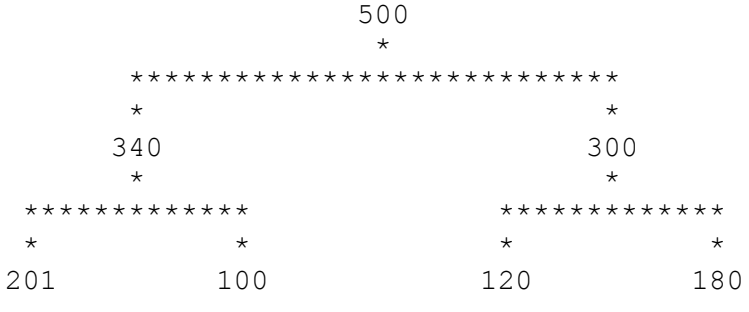
Prvi način koristi se u skripti iz predmeta ASP. Ja ću ovdje koristiti drugi način, ne zato da Vas zbunim, već zato što je drugi način daleko logičniji - u C-u sva polja počinju sa indeksom 0! Evo prikaza mogućih indeksacija, i gore prikazanog heapa:

Indeks polja	Elementi po prvoj indeksaciji	Elementi po drugoj indeksaciji
0	???	505
1	505	500
2	500	300
3	300	201
4	201	340
5	340	120
6	120	180
7	180	nije element polja

Zamislite da imate stvoren heap, kao onaj naš gore, i na vrh mu, umjesto elementa 505 stavite element 100. Ta struktura više ne predstavlja heap, ali kako možemo stvar popraviti? Pogledajmo djecu od tog čvora. To su 500 i 300. Kako roditelj mora biti veći od oba djeteta, pogledajmo koje je dijete veće => 500. Zamijenimo sada elemente 500 i 100. Izmjenu smo napravili u lijevom podstablu i dobili ovo:



Da li se je ovime narušilo desno podstablo? Nije! Zato ga u nastavku više nećemo niti promatrati. Sada pogledajmo čvor u koji smo stavili element 100. Vidimo da je on manji od djece, pa ovo nije gomila. Što treba učiniti da se stvar popravi? Pa isto što smo učinili i maloprije! Nađimo veće dijete => 340 i zamijenimo ga sa elementom 100!



I evo ga, nakon što smo narušili korijen heapa, rekurzivnim (*ili iterativnim? zna li netko razliku?*) ponavljanjem opisanog postupka opet smo dobili heap! Ovaj postupak koristiti ćemo prilikom stvaranja gomile. Evo kako. Pogledajmo jedan čvor stabla R koji ima djecu LD i DD. Kada će taj čvor zadovoljavati strukturu heap sa relacijom *veći od*? Samo ako vrijedi R>LD, R>DD. Ako ovo ne vrijedi, možemo izreći jednu pretpostavku (a pojasniti ću i kada ona vrijedi). Recimo ovako: ako moj čvor ne zadovoljava strukturu heap, smatrati ćemo da je to zato, što je u ispravnoj strukturi heap (koju je taj čvor tvorio) korijen zamijenjen neadekvatnim elementom (kao što sam ja u gornjem primjeru zamijenio broj 505 sa 100). Ako je to tako, tada ćemo provesti opisani algoritam koji će ispraviti tu pogrešku i ponovno realizirati strukturu heap! A kada dana pretpostavka vrijedi? Neka je zadano binarno stablo koje nije uređeno kao heap. Zamislite na tren da gornja sličica ne prikazuje heap. Pitam Vas da li čvor sa brojem 180 čini strukturu heap? Pa očito čini, jer nema djece! Isto vrijedi i za sve skroz donje elemente stabla - ili listove, ako Vas to baš veseli. Dakle, i 120, 100, 201 ne narušavaju definiciju heapa. Idemo dalje. Što je sa čvorom 300? E on bi mogao narušiti heap jer ima djece. No ono što sigurno znamo je da su njegova djeca *ispravni heap-ovi*! I tada možemo primijeniti gornju pretpostavku da 300 "narušava" heap i provesti gornji algoritam koji će to popraviti. Nakon toga sada i 300 i njegova djeca čine ispravan heap. Idemo dalje. Što je sa čvorom sa vrijednošću 340? Vrijedi potpuno ista priča kao za čvor 300 - ako nije heap, mogu ga popraviti jer mu djeca jesu heap! I konačno se dižemo još razinu iznad do broja 500. Je li on zadovoljava heap? Ako da, gotovi smo, ako ne, djeca mu jesu heapovi pa možemo pogrešku ispraviti! I na ovaj način iz običnog binarnog stabla možemo stvoriti heap. Nadam se da ste uočili dvije bitne stvari: krećemo se ***od desna u lijevo***, i ***od predzadnje razine prema prvoj***! Zadnju razinu nema potrebe provjeravati jer to jesu heapovi budući da zadnja razina (očito) nema djece.

Još par detalja prije samog algoritma. Budući da koristimo polje sa indeksima od 0, rekli smo da vrijedi: ako je roditelj na indeksu n, djeca su na 2*n+1 i 2*n+2. Idemo vidjeti i obrat: ako je dijete na indeksu n, gdje mu je roditelj? Može se pokazati da mu je roditelj na indeksu (n-1)/2 pri čemu je dijeljenje cjelobrojno! Formulu možemo provjeriti na primjeru:

dijete	1	2	3	4	5	6	7	8	9	10
roditelj	0	0	1	1	2	2	3	3	4	4

Funkciju koja realizira ispravljanje krivog korijena heap-a nazvati ćemo Podesi. Evo realizacije:

```
void Podesi( int *array, int n, int i ) {
 int j;
 int t;

 // Izracunaj indeks lijevog djeteta
 j = 2*i+1;
 // Zapamti vrijednost korjena (tj. element koji kvari strukturu)
 t = array[i];
 // Tako dugo dok se nismo spustili do dna strukture, radi ...
 while( j < n ) {
  // ako desno dijete postoji, i ako je veće od lijevog, tada
  // neka j postane indeks desnog djeteta
  if( j < n-1 && array[j] < array[j+1] ) j++;
  // ako veće dijete nije veće od roditelja, prekid!
  if( t >= array[j] ) break;
  // Inače treba popraviti strukturu. Zato u roditelja kopiramo veće dijete,
  // a u veće dijete bismo trebali roditelja: t, no ako će i ispod trebati
  // mjenjati, onda smo bezveze kopirali pa to ne radimo već pamtimo da je to t.
  // Ujedno proglašavamo veće dijete roditeljem, i računamo indeks njegove
djece.
  array[i] = array[j]; i=j; j = 2*j+1;
 }
 // sada konačno stavljamo ovu zapamćenu vrijednost na mjesto bivšeg većeg
djeteta.
 array[i]=t;
}
```

"array" je pokazivač na polje, "n" je broj elemenata polja a "i" je indeks elementa (korijena) koji kvari strukturu heap. Idemo sada realizirati i funkciju koja pozivanjem gornje na već opisan način stvara heap. Rekli smo da krećemo od predzadnje razine. Ako polje ima n elemenata, krajnji desni element na predzadnjoj razini nalazi se na indeksu (n-2)/2; naime, uz n elemenata, indeksi idu od 0 do n-1. Roditelj "n-1"-vog djeteta je ("n-1"-1)/2 = (n-2)/2. I od tuda se penjemo prema gore po svim čvorovima krenuvši u lijevu stranu. No kada je stvar pohranjena u polje, to jednostavno znači da sa indeksima idemo od (n-2)/2 do nule! Evo funkcije:

```
void StvoriGomilu( int *array, int n ) {
 int i;

 for( i=(n-2)/2; i>=0; i-- )
  Podesi(array,n,i);
}
```

Mislim da funkciju ne treba dodatno komentirati. No kako ovo sve možemo iskoristiti za sort? Pa jednostavno: gomila na vrhu uvijek ima najveći element. Što mi radimo? Zamijenimo njega i zadnjeg elementa polja. Zatim kažemo da nam je polje za jedan kraće, a budući da smo zadnji element polja stavili na vrh heap-a, još i pozovemo funkcije Podesi da popravi nastalu neispravnost, ukoliko ona postoji. No nakon njenog rada na vrhu je opet najveći element iz tog skraćenog polja! Njega sada frknemo na predzadnje mjesto, a onoga sa predzadnjeg mjesta na vrh. Opet duljinu polja skratimo za jedan, a budući da je opet struktura narušena, zovemo Podesi... i tako sve dok ne dođemo do duljine polja od jednog elementa, kada prekidamo proces jer je polje od jednog elementa očito sortirano. U kod pretočeno, stvar izgleda ovako:

```
void HeapSort( int *array, int n ) {
 int i;
 int t;

 StvoriGomilu(array,n);
 for(i=n; i>=2; i-- ) {
  t = array[i-1]; array[i-1] = array[0]; array[0] = t;
  Podesi( array, i-1, 0 );
 }
}
```

Očito je što se događa. "i" nam glumi duljinu polja; petlja se vrti dok je "i" veći od 1. Ako je polje dugačko "i" elemenata, tada je zadnji na indeksu "i-1". Njega zamijenimo sa prvim, i popravimo strukturu, i tako n puta. Budući da se funkcija popravi spušta po razinama, ako imamo n elemenata, razina je $(\log_2 n)+1$, te je ugrubo broj pretumbavanja elemenata proporcionalan na $n*\log_2 n$, što je puno puno bolje od kvadratnih ovisnosti za iole veće brojeve.

SHELL SORT

Već ste vidjeli kako promjena kuta gledanja na određen problem može uroditi različitim rješenjima istoga. Prvo smo na polje brojeva gledali kao na - polje brojeva i dobili smo kvadratne ovisnosti. Zatim smo na polje brojeva pogledali malo drugačije - kao na heap, i već smo dobili n puta logaritamsku ovisnost. Idemo sada na polje gledati kao na - polje! Da, znam. Ništa novoga. Ali ovaj puta za ideju za sortiranje nećemo pitati maloga Ivicu. A, ne. Sada ćemo se obratiti Donaldu Shellu! A on će nam reći slijedeće:

k sortirano polje *je ono polje za koje vrijedi da je* ***polje[i] <= polje[i+k]****, i to za svaki* ***i*** *(takav da indeks i+k ima smisla).*

Ako je polje ***k*** *sortirano, i dodatno se* ***t*** *sortira (****t****<****k****), polje tada ostaje i dalje i* ***k*** *sortirano. (dakle, ako polje 5-sortiramo, i onda ga 2-sortiramo, polje će biti i 5 i 2 sortirano).*

Polje brojeva treba sortirati počev sa k1-sortom, pa k2-sortom, ..., do 1-sorta, k1>k2>...>1.

Ono što mi u klasičnom smislu zovemo sortiranim polje, u ovoj se notaciji može opisati kao 1-sortirano polje. Ideja shell sorta je da se polje ne sortira odmah 1-sortom, već da se krene od većih sortova prema 1-sortu. Slijed ovih brojeva može biti različit, i ovisno o njemu, razlikuje se i brzina samog sorta. U nastavku će biti prikazan sort pomoću slijeda brojeva n/2, n/4, n/8, n/16, ..., 1, pri čemu je n broj elemenata polja.

Funkcija je slijedeća:

```
void ShellSort( int *array, int n ) {
 int i,j,t,step;

 for( step=n/2; step>0; step/=2 ) {
  for( i = step; i < n; i++ ) {
   t = array[i];
   for( j=i; j >= step; j -= step ) {
    if( t < array[j-step] ) array[j] = array[j-step];
    else break;
   }
   array[j] = t;
  }
 }
}
```

Evo primjera za polje brojeva {15, 24, 11, 17, 36, 13, 25, 62, 23, 18, 33, 51, 22}

```
Početno stanje:      15, 24, 11, 17, 36, 13, 25, 62, 23, 18, 33, 51, 22

Nakon sorta sa k= 6: 15, 24, 11, 17, 33, 13, 22, 62, 23, 18, 36, 51, 25
Nakon sorta sa k= 3: 15, 24, 11, 17, 33, 13, 18, 36, 23, 22, 62, 51, 25
Nakon sorta sa k= 1: 11, 13, 15, 17, 18, 22, 23, 24, 25, 33, 36, 51, 62
```

Sada je nadam se postalo potpuno jasno što znači da je polje k-sortirano, no ako nije, evo ga još jednom pa provjerite na gornjem polju: polje je k sortirano, ako vrijedi polje[1] < polje[1+k],polje[2] < polje[2+k],polje[3] < polje[3+k], ... sve do kraja polja (uz pretpostavljenu indeksaciju polja od 1 na više).

Kako se dani algoritam može realizirati? Pogledajmo prvo kako ćemo realizirati neki zadani k-sort, pri čemu je k fiksan broj. Možemo krenuti ovako: usporedi polje[0] sa polje[0+k] i ako treba, zamjeni ih. Zatim pogledaj polje[1] i polje[1+k] pa ako treba, zamjeni ih... itd. No što se događa kada dođemo do usporedbe elemenata polje[0+k] i polje[0+2*k], i recimo da su elementi takvi da ih treba zamijeniti? Tada je moguće da je element koji se je nalazio na polje[0+2*k] i koji je sada došao na mjesto polje[0+k] manji od elementa polje[0]! A tada je narušena k sortiranost polja! Evo primjera o čemu govorim:

```
Zadan je k=4 i slijedeće polje:
40, 41, 42, 43, 44, 45, 46, 47, 20, 21, 23, 28
                 *                  *
                 ****************** - nije dobro pa zamjenjujemo 44 i 20
40, 41, 42, 43, 20, 45, 46, 47, 44, 21, 23, 28
 *                 *
 ****************** - sada smo narušili ovaj dio koji je bio u redu!
```

Da bismo spriječili da nam se događaju ovakva narušavanja, moramo nakon što napravimo zamjenu elemenata polje[4] i polje[8] provjeriti u jesmo li narušili odnos polje[0] i polje[4], pa ako jesmo, treba i ta dva elementa zamijeniti (i kada bi polje[0] imalo svoga prethodnika za korak k ispred, trebalo bi i tu provjeriti odnos pa i to ispraviti, itd. sve dok ima prethodnika). No uočite da ukoliko elementi polje[4] i polje[0] ne bi narušavali sort, tada one elemente ispred njih ne bi niti trebalo ispitivati jer smo njih već prije složili ispravno. Općenito govoreći,

- *kada ispitujemo elemente polje[i+(n+1)*k] i polje[i+n*k] i napravimo zamjenu, treba redom ispitivati i elemente polje[i+n*k] i polje[i+(n-1)*k], pa ako smo tu napravili zamjenu, treba*

*ispitati elemente polje[i+(n-1)*k] i polje[i+(n-2)*k], ... Ukoliko se pokaže da polje[i+(n-j)*k] i polje[i+(n-j-1)*k] zadovoljavaju k-sort, daljnje ispitivanje unatrag možemo prekinuti*

Nadalje, ispitivati polje možemo od početka, pa tada treba ispitivati sve elemente od *0*-tog to *n-1-k*-tog, jer kada smo na indeksu *n-1-k*, tada se očito ispituju elementi polje[*n-1-k*] i polje[*n-1-k+k*]=polje[*n-1*] što je zadnji element polja. No priču možemo okrenuti i na drugu stranu, pa umjesto ispitivanja elemenata polje[i] i polje[i+k] ići ispitivati elemente polje[i] i polje[i-k]. Ako radimo tako, tada indeks *i* treba ići od *k* do *n-1*. Tada radimo unazadno ispitivanje.

Dalje, uzmimo da smo nakon ispitivanja elemenata x1=polje[i+(n+1)*k] i x2=polje[i+n*k] došli do zaključka da ih treba zamijeniti. Radimo zamjenu: polje[i+(n+1)*k]=x2 i polje[i+n*k]=x1. Zatim, budući da smo napravili zamjenu, trebamo ispitati i elemente polje[i+n*k] (=x1) i x3=polje[i+(n-1)*k], i elemente opet treba zamijeniti! Radimo zamjenu: polje[i+n*k]=x3 i polje[i+(n-1)*k]=x1. Sada opet moramo ispitati odnos elemenata na koraku ispred, pa uspoređujemo polje[i+(n-1)*k] (=x1) i x4=polje[i+(n-2)*k]. Neka ova dva elementa zadovoljavaju k-sort, pa tu možemo prekinuti povratke unazad. Pogledajmo što se je događalo sa pojedinim elementima:

Prije usporedbi i zamjena	Nakon prve usporedbe i zamjene	Nakon druge usporedbe i zamjene

`polje[i+(n-2)*k]=x4`
`...`
`polje[i+(n-1)*k]=x3`
`...`
`polje[i+( n )*k]=x2`
`...`
`polje[i+(n+1)*k]=x1`
`...`
	`polje[i+(n-2)*k]=x4`
	`...`
	`polje[i+(n-1)*k]=x3`
	`...`
	`polje[i+( n )*k]`**`=x1`**
	`...`
	`polje[i+(n+1)*k]`**`=x2`**
	`...`
		`polje[i+(n-2)*k]=x4`
		`...`
		`polje[i+(n-1)*k]`**`=x1`**
		`...`
		`polje[i+( n )*k]`**`=x3`**
		`...`
		`polje[i+(n+1)*k]=x2`
		`...`

Vidimo da smo element x1 najprije kopirali sa indeksa *i+(n+1)*k* na indeks *i+n*k*, pa zatim sa tog indeksa na indeks *i+(n-1)*k*. Ovime smo radili jedno kopiranje viška! Da smo se vratili još jedan korak unatrag, radili bismo još jedno kopiranje viška! Da se ovo izbjegne, možemo postupiti na slijedeći način:

- Prije no što počnemo ispitivati, vrijednost elementa *polje[i+(n+1)*k]* zapamtimo u privremenoj varijabli *t*. Zatim radimo ispitivanje između vrijednosti *t* i *polje[i+n*k]*, pa ako ih treba zamijeniti, vrijednost elementa *polje[i+n*k]* kopiramo u element *polje[i+(n+1)*k]*, ali *t* ne kopiramo u *polje[i+n*k]*, već pamtimo da tu mora doći *t*! Sada, budući da smo napravili izmjenu, moramo se vratiti korak unatrag i ispitati odnos elemenata *polje[i+n*k]* i *polje[i+(n-1)*k]*. No kako *t* nismo prekopirali u *polje[i+n*k]*, raditi ćemo usporedbu elemenata *t* i *polje[i+(n-1)*k]*! Recimo da je potrebna zamjena, pa vrijednost elementa polje[i+(n-1)*k] kopiramo u element polje[i+n*k], ali opet ne radimo kopiranje elementa *t* u element *polje[i+(n-1)*k]*, nego sada opet idemo korak unatrag i ispitujemo odnos elemenata *polje[i+(n-1)*k]* i *polje[i+(n-2)*k]*, odnosno *t* i *polje[i+(n-2)*k]* jer *i* nismo kopirali u *polje[i+(n-1)*k]*. Neka je sada sve u redu, pa prekidamo povratke unatrag. Jedino što je još ostalo je konačno element pohranjen u *t* iskopirati na njegovo pravo mjesto, a to je *polje[i+(n-1)*k]*, jer smo tu stali sa povratcima.

Uočite da na ovaj način izbjegavamo nepotrebno kopiranje onog elementa koji putuje prema naprijed sve do trenutka kada nađemo pravo mjesto gdje ga treba staviti.

Kako je algoritam doista realiziran? Pa ako se sada vratite na tijelo funkcije, mislim da će vam sve biti jasno. U funkciji se koriste tri for petlje. Vanjska petlja odbrojava korak sortiranja, i kao što je već rečeno, kreće sa n/2, pa sa n/4, itd. sve do 1. Srednja petlja odbrojava indeks i počev od k-tog (tj. od koraka sorta) pa sve do n-1. I konačno unutarnja petlja služi za unazadno ispitivanje elemenata (petlja se produžuje tako dugo se obavlja kopiranje, ili dok se ne stigne do indeksa manjeg od koraka, jer kako se radi unazadno ispitivanje, j-k nema smisla za j<k, jer negativni elementi ne postoje!). Cjelobrojna varijabla t služi kao privremena pohrana elementa koji će se vraćati prema početku polja (budući da je riječ o polju cijelih brojeva).

Efikasnost algoritma uvelike ovisi o slijedu koraka koji se primjenjuju prilikom sortiranja. Jedan od poznatijih je Hibbardov slijed brojeva, koji napisan naopačke glasi: 1,3,7,15,31,...,2^i-1, koji u najgorem slučaju ima $O(n^{3/2})$, a prosječno $O(n^{5/4})$. Evo jedne moguće realizacije ShellSort-a koji koristi taj slijed:

```
void ShellSortHibb( int *array, int n ) {
 int i,j,t,step,s;

 if( n < 2 ) return;
 s = log(n)/log(2);
 s = pow( 2., (double)s );
 for( ; s>1; s/=2 ) {
  step=s-1;
  for( i = step; i < n; i++ ) {
   t = array[i];
   for( j=i; j >= step; j -= step ) {
    if( t < array[j-step] ) array[j] = array[j-step];
    else break;
   }
   array[j] = t;
  }

 }
}
```

Postoje još neki nizovi, od kojih je za sada najbolji Sedgwickov slijed {1,5,19,41,109,...} koji se formulom može izraziti kao naizmjenično uzimanje brojeva koje generira $9*4^i-9*2^i+1$ te 4^i-3*2^i+1.

SORTIRANJE RAZBIJANJEM. MERGE SORT.

Do sada smo vidjeli razne varijante sortova. Za sve smo njih zaključili: što je više elemenata, to sort duže traje; no ovisnost nije linearna, i to nam otvara mjesta novoj ideji iskazanoj poznatim Merge sort algoritmom. Ideja je slijedeća.

- Neka imamo polje od n elemenata. Da bi ga sortirali nekim Bubble sortom, treba nam, okvirno govoreći uz složenost sorta od n^2, upravo n^2 usporedbi. No ako sada polje razbijemo na dva podpolja, svako sortiramo posebno, i zatim slijednim uzimanjem elemenata po veličini iz jednog ili drugog podpolja (ovisno koji je manji) složimo novo polje, trebati će nam: $(n/2)^2 + (n/2)^2 + n$ usporedbi, što je $n^2/4+n$, što je sigurno manje od n^2 za veće n (ovaj podatak vrijedi uz pretpostavku da manja polja sortiramo bubble sortom!). Dakle, zbog toga što za sortiranje nemamo linearne ovisnosti, isplati se polje razbiti u manje dijelove, i onda stvar sortirati posebno, pa zatim sortirana polja spojiti u jedno, novo polje.

Koji je zaključak ovoga? Dobili smo na brzini, ma kojim algoritmom sortirali manja podpolja. No, sigurno ste uočili da sam spominjao novo polje. Da, točno. Cijena koju moramo platiti je u memoriji! Moramo stvoriti još jedno pomoćno polje! Nadalje, da me ne biste krivo shvatili, manja podpolja se ne moraju nužno sortirati nekim drugim algoritmom. Merge sort može se primjenjivati i rekurzivno! A ako to iskoristimo, još ćemo dobiti na brzini. Naime, ako polja rekurzivno dijelimo po pola, možemo zamisliti da se time stvara binarno stablo sa $(\log_2 n)+1$ razina. Na pojedinoj razini složenost je linearna O(n), pa je ukupna složenost $O(n*\log_2 n)$.

Pogledajmo najprije kako ćemo dva sortirana podpolja složiti u novo sortirano polje. Uzmimo primjer:

korak	početna polja: polje1: 1,5,9,13,19 polje2: 2,3,7,16,20		novo polje
1.	**1**,5,9,13,19 2,3,7,16,20	uzimamo 1	1
2.	5,9,13,19	uzimamo 2	1,2

	2,3,7,16,20		
3.	5,9,13,19 **3**,7,16,20	uzimamo 3	1,2,3
4.	**5**,9,13,19 7,16,20	uzimamo 5	1,2,3,5
5.	9,13,19 **7**,16,20	uzimamo 7	1,2,3,5,7
6.	**9**,13,19 16,20	uzimamo 9	1,2,3,5,7,9
7.	**13**,19 16,20	uzimamo 13	1,2,3,5,7,9,13
8.	19 **16**,20	uzimamo 16	1,2,3,5,7,9,13,16
9.	**19** 20	uzimamo 19	1,2,3,5,7,9,13,16,19
10.	**20**	uzimamo 20	1,2,3,5,7,9,13,16,19,20

Dakle, uz dva polja redom izvlačimo uvijek manji broj i njega dodajemo u novo polje. Izvlačenje brojeva radimo tako dugo dok ima raspoloživih brojeva. Evo i programske realizacije algoritma. Funkcija će dobiti pokazivač na polje koje sadrži podpolja, pokazivač na pomoćno polje, lijevi graničnik (prvi element prvog podpolja), desni graničnik(prvi element drugog podpolja, s time da se prvo podpolje proteže do tog elementa, ne uključujući ga), i zadnji element drugog podpolja, tj. desni kraj. Dodatno, sortirani se niz neće u pomoćno polje kopirati od početka, već od iste pozicije na kojoj počinje u originalnom polju (za potrebe sortiranja, inače nepotrebno).

```
void Merge( int *array, int *tmp, int lpos, int rpos, int rend ) {
 int i, lend, n, k;

 lend = rpos-1;
 k = lpos;

 n = rend - lpos + 1;
 // Sve dok jedno od polja ne ostane bez elemenata, radi ...
 while( lpos <= lend && rpos <= rend ) {
  // ako je element lijevog podpolja manji od desnog, uzmi njega,
  if( array[lpos] < array[rpos] )
   tmp[k++] = array[lpos++];
  // inace uzmi element desnog podpolja
  else
   tmp[k++] = array[rpos++];
 }
 // Sada je sigurno jedno od polja ostalo bez elemenata, pa
 // ako u lijevom jos ima nesto, to sve prekopiraj ...
 while( lpos <= lend ) {
   tmp[k++] = array[lpos++];
 }
 // ili mozda u desnom ima jos nesto pa to sve prekopiraj
 // treba uočiti da se neće izvršiti obje while petlje jer je
 // jedno polje sigurno ostalo bez elemenata!
 while( rpos <= rend ) {
   tmp[k++] = array[rpos++];
 }
 // i konačno prekopiraj tih n elemenata natrag u originalno polje
 for( i = 0; i < n; i++, rend-- )
  array[rend] = tmp[rend];
}
```

Kako sam već rekao, za sortiranje podpolja može se uzeti bilo koji algoritam, pa i sam MergeSort. Da ovo demonstriramo, evo i rekurzivne funkcije koja obavlja sortiranje baš po ideji rekurzivnog Merge Sorta. "left" je lijevi (prvi) indeks elementa podpolja, a "right" je desni (zadnji) indeks elementa podpolja.

```
void MergeRecursive( int *array, int *tmp, int left, int right ) {
 int middle;

 // ako polje ima više od jednog elementa, radi ...
 if( left < right ) {
  // nađi sredinu
  middle = (left+right)/2;
  // rekurzivno sortiraj podpolje nastalo od prvog indeksa do srednjeg
  MergeRecursive( array, tmp, left, middle );
  // rekurzivno sortiraj podpolje nastalo od srednjeg indeksa + 1 do zadnjeg
indeksa
  MergeRecursive( array, tmp, middle + 1, right );
  // i spoji ta dva podpolja
  Merge( array, tmp, left, middle+1, right );
 }
}
```

I još nam treba funkcija koja će alocirati pomoćno polje, i pozvati rekurzivni merge sort:

```
void MergeSort( int *array, int n ) {
 int *tmp;

 tmp = (int*)malloc(n*sizeof(int));
 if( tmp == NULL ) return; // greska!
 MergeRecursive( array, tmp, 0, n-1 );
 free( tmp );
}
```

Ha? Što kažete? Sve jasno? Pogledajmo kako algoritam radi na polju {9,8,7,5,2}. Funkcija MergeSort stvori pomoćno polje, i pozove funkciju MergeRecursive sa granicama 0 do 4 (jer polje ima 5 elemenata). Od tuda možemo dalje pratiti rad algoritma; svaki puta kada se ispiše "Pozvan rekurzivni sort dijela x do y" znači da je pozvana funkcija MergeRecursive sa granicama x do y; slijedeći redak će tada sadržavati i obavijest koji su to elementi. Nakon što jedna instanca MergeRecursive funkcije završi sa sortiranjem, ispisati će se rezultat "Rezultat sortiranja dijela x do y:", te izgled cijelog polja u tom trenutku. Pogledajte!

```
Pozvan rekurzivni sort dijela 0 do 4.
Traži se sortiranje elemenata:  9, 8, 7, 5, 2
 *Pozvan rekurzivni sort dijela 0 do 2.
 *Traži se sortiranje elemenata:  9, 8, 7
 * *Pozvan rekurzivni sort dijela 0 do 1.
 * *Traži se sortiranje elemenata:  9, 8
 * * *Pozvan rekurzivni sort dijela 0 do 0.
 * * *Traži se sortiranje elemenata:  9
 * * *Rezultat sortiranja dijela 0 do 0:  9
 * * *Cijelo polje:  9  8, 7, 5, 2,
 * * ----------------------------------------
 * * *Pozvan rekurzivni sort dijela 1 do 1.
 * * *Traži se sortiranje elemenata:  8
 * * *Rezultat sortiranja dijela 1 do 1:  8
 * * *Cijelo polje:  9, 8  7, 5, 2,
 * *Rezultat sortiranja dijela 0 do 1:  8, 9
 * *Cijelo polje:  8, 9  7, 5, 2,
 * ----------------------------------------
 * *Pozvan rekurzivni sort dijela 2 do 2.
 * *Traži se sortiranje elemenata:  7
 * *Rezultat sortiranja dijela 2 do 2:  7
 * *Cijelo polje:  8, 9, 7  5, 2,
 *Rezultat sortiranja dijela 0 do 2:  7, 8, 9
```

```
 *Cijelo polje:  7, 8, 9  5, 2,
 ----------------------------------------
 *Pozvan rekurzivni sort dijela 3 do 4.
 *Traži se sortiranje elemenata:  5, 2
 * *Pozvan rekurzivni sort dijela 3 do 3.
 * *Traži se sortiranje elemenata:  5
 * *Rezultat sortiranja dijela 3 do 3:  5
 * *Cijelo polje:  7, 8, 9, 5  2,
 * ----------------------------------------
 * *Pozvan rekurzivni sort dijela 4 do 4.
 * *Traži se sortiranje elemenata:  2
 * *Rezultat sortiranja dijela 4 do 4:  2
 * *Cijelo polje:  7, 8, 9, 5, 2
 *Rezultat sortiranja dijela 3 do 4:  2, 5
 *Cijelo polje:  7, 8, 9, 2, 5
Rezultat sortiranja dijela 0 do 4:  2, 5, 7, 8, 9
Cijelo polje:  2, 5, 7, 8, 9
```

Uočite da svaki poziv funkcije MergeRecursive rezultira sa dva nova poziva iste funkcije, kao što se i može očekivati iz koda funkcije.

SORTIRANJE UBACIVANJEM

Možda sam ovaj sort trebao obraditi malo ranije, no budući da nije sort kojega bi se čovjek odmah sjetio, a i nije baš nešto, eto ga na ovom mjestu. Moram priznati, ideja je također trivijalna, a baš to je ono čega se čovjek najčešće ne sjeti. Evo ideje:

- polje od jednog elementa je sortirano
- neka imamo polje od n+1 elemenata, pri čemu je prvih n sortirano, a n+1-vi element narušava poredak; tada će i svih n+1 elemenata biti sortirano, ako izvedemo slijedeći algoritam:
  zapamti *n+1*-vi element u pomoćnoj varijabli
  budući da je *n+1*-vi element manji od *n*-tog, pomakni *n*-ti element na *n+1*-vu lokaciju, te sada ispitaj odnos zapamćenog elementa sa *n-1*-vim elementom; ako je on veći od zapamćenog, pomakni *n-1*-vi element na *n*-tu lokaciju, pa ispitaj odnos *n-2*-og elementa sa zapamćenim; ako je on veći,... shvatili ste. Onog trena kada *n-i*-ti element postane manji od zapamćenog, pravi je tren da zapamćeni element prekopiramo na *n-i+1*-vu lokaciju; i time je polje opet sortirano.
- Drugim riječima, kreni od kraja i sve brojeve koji su veći od zapamćenog gurni za jedno mjesto u desno; u nastalu rupu prekopiraj zapamćeni element.
- Ponovi gore opisani postupak za n=1 do n=broj_elemenata_polja-2 (uz gore navedeno značenje varijable n)

Evo i primjera na našem polju {9,8,7,5,2}

```
Početno stanje u prolazu 1: 9, 8, 7, 5, 2

Pamtim element na indeksu 2, a to je 8.

Guram 1. element na 2. poziciju (9==>8).
Polje izgleda: 9, 9, 7, 5, 2
Na lokaciju 1 stavljam zapamćeni element 8
Završno stanje u prolazu 1: 8, 9, 7, 5, 2
```

```
Početno stanje u prolazu 2: 8, 9, 7, 5, 2

Pamtim element na indeksu 3, a to je 7.

Guram 2. element na 3. poziciju (9==>7).
```

```
Polje izgleda: 8, 9, 9, 5, 2
Guram 1. element na 2. poziciju (8==>9).
Polje izgleda: 8, 8, 9, 5, 2
Na lokaciju 1 stavljam zapamćeni element 7
Završno stanje u prolazu 2: 7, 8, 9, 5, 2
```

```
Početno stanje u prolazu 3: 7, 8, 9, 5, 2

Pamtim element na indeksu 4, a to je 5.

Guram 3. element na 4. poziciju (9==>5).
Polje izgleda: 7, 8, 9, 9, 2
Guram 2. element na 3. poziciju (8==>9).
Polje izgleda: 7, 8, 8, 9, 2
Guram 1. element na 2. poziciju (7==>8).
Polje izgleda: 7, 7, 8, 9, 2
Na lokaciju 1 stavljam zapamćeni element 5
Završno stanje u prolazu 3: 5, 7, 8, 9, 2
```

```
Početno stanje u prolazu 4: 5, 7, 8, 9, 2

Pamtim element na indeksu 5, a to je 2.

Guram 4. element na 5. poziciju (9==>2).
Polje izgleda: 5, 7, 8, 9, 9
Guram 3. element na 4. poziciju (8==>9).
Polje izgleda: 5, 7, 8, 8, 9
Guram 2. element na 3. poziciju (7==>8).
Polje izgleda: 5, 7, 7, 8, 9
Guram 1. element na 2. poziciju (5==>7).
Polje izgleda: 5, 5, 7, 8, 9
Na lokaciju 1 stavljam zapamćeni element 2
Završno stanje u prolazu 4: 2, 5, 7, 8, 9
```

A evo i koda funkcije:

```
void UbaciSort( int *array, int n ) {
 int i,j;
 int t;

 for( i = 1; i < n; i++ ) {
  t = array[i];
  for( j=i; j>0 && array[j-1]>t; j-- ) {
   array[j]=array[j-1];
  }
  array[j] = t;
 }
}
```

Složenost funkcije je opet kvadratna, i to broj usporedbi je n*(n-1)/2, što je ujedno jednako i broju kopiranja elemenata u najgorem slučaju!

Sada jedna loša vijest: QuickSort biti će obrađen u jednoj od slijedećih prigoda...

INDIREKTNO SORTIRANJE

Na našu veliku žalost, u svijetu računala, valjda jedina stvar koju ne treba sortirati je ono što se najlakše sortira: brojeve. Gotovo je uvijek riječ o nekakvim strukturama koje treba sortirati prema zadanim elementima iste. Dodatno, svaka ta struktura ima i svoju veličinu koja je nerijetko poveća. Postavlja se pitanje kako tada efikasno sortirati jedno polje takvih struktura? Uzmimo za primjer da radimo sa strukturama veličine 1KB (jedan kilobajt), i imamo polje od 1000 struktura. Brojke uopće nisu velike. Pogledajmo koliko memorije prebacimo zamjenom dva elementa polja?

1. `kopiraj element A u pomoćnu varijablu ==> jedanput se kopira veličina elementa`
2. `kopiraj element B u element A ==> jedanput se kopira veličina elementa`
3. `kopiraj pomoćnu varijablu u element B ==> jedanput se kopira veličina elementa`

Suma kaže po jednoj zamjeni prekopiramo *tri veličine danog objekta*!

Pogledajmo sada našu prvu funkciju za sortiranje: Sort0! Ona radi (n-1)*(n-1) provjera, što uz n=1000 iznosi 998001 (malo manje od milijun). Isto tako, u najgorem slučaju funkcija radi n*(n-1)/2 zamjena elemenata, što u našem slučaju iznosi 499500 zamjena. Za svaku se zamjenu prebace 3 veličine objekta koji se zamjenjuje, pa kod nas imamo 1534464000 bajtova (dobro vidite; uz veličinu objekta od 1KB mi prebacimo 1463,4 MB memorije - skoro gigabajt i pol)!

Funkcija Sort01 ima samo nešto manji broj usporedbi: 499500 usporedbi, te broj zamjena isti kao i prethodna funkcija: 499500 zamjena. Dakle, i ona okrene 1463,4 MB memorije.

Pogledajmo funkciju BubbleSort1. Ona ima broj usporedbi jednak prethodnoj funkciji: 499500 usporedbi. No koliki je njezin broj zamjena elemenata? n-1! Samo n-1. Ona će dakle napraviti 999 zamjena, ili okrenuti 3068928 bajtova ili 2,9MB memorije!

Pa sada vi meni recite da veličina objekta nije važna za sortiranje! A na 32-bitnim mašinama jednom se instrukcijom u općem slučaju mogu prekopirati maksimalno 4 bajta! Dobro, sada ćete vi reći da su današnje mašine jako brze i uz cca jedan ciklus od 10 ns kopiranje svih gigabajt i pol iz gornjeg primjera traje oko 3,84 sekunde. A onda ću ja vama reći realna brojka takovih struktura u polju nije 1000 nego milion!! Pa da vas čujem onda?

No, nemojte sada misliti da broj usporedbi nije važan za algoritam. Dakako da je. No ono što sam htio istaknuti je da i dva algoritma sa "istim" brojem usporedbi mogu imati jaaaaakkko različita vremena sortiranja upravo zbog različitog broja kopiranja.

No, može li se ikako doskočiti ovom problemu? Naravno da može! Recimo da imate polje struktura koje sadrže podatke o nekoj osobi: ime, prezime, jmbg, adresu, telefon, fax, e-mail, web adresu, datum rođenja, naziv tvrtke u kojoj radi, broj telefona gdje ga se može tamo dobiti, boju kose, boju očiju, popis svih bolesti od kojih je bolova, ... Da ne duljim, zamislite da skupljate ono što skuplja CIA ili FBI. I sada trebate tih par desetaka milijuna (ili više?) podataka o amerima posortirati po, recimo jmbg-u (ili kako oni već numeriraju ljude). Jedna struktura u koju bi stali svi gore navedeni podaci sigurno bi bila jako velika, i raditi sa takvim strukturama bilo bi, pa recimo to ovako - jako nepreporučljivo. No zato možemo napraviti slijedeće: možemo stvoriti polje struktura koje sadrže samo onaj podatak koji sortiramo (jmbg) i nekakav pokazivač na izvornu strukturu otkuda smo uzeli taj jmbg. Ova nova struktura biti će daleko manja, i brže ćemo issortirati to novo polje. Jednom kada smo završili sa sortiranjem, originalno polje možemo preurediti u sortirano u linearnom vremenu - jer sada točno znamo gdje moramo staviti koju strukturu, budući da nam je malo polje sortirano, a svaki njegov element sadrži pokazivač na originalnu strukturu! Ovdje opisan postupak zove se, nikada nećete pogoditi - indirektno sortiranje.

OPĆENITE FUNKCIJE ZA SORTIRANJE

Do sada smo govorili isključivo o poljima cijelih brojeva. Postavlja se pitanje može li se napraviti funkcija kojoj bi bilo svejedno što sortira, tako dugo dok je to pohranjeno kao polje? Brzo! Otiđite na početak ovog teksta i nađite prvi algoritam malog Ivice. Vidite li negdje spominjanje brojeva? Ne? A drugi? A treći? Baš nigdje? Da, i mislio sam si tako. Kada smo govorili o algoritmima za sortiranje, nije nam bilo bitno što sortiramo, već nam je bilo bitno KAKO sortiramo. Ono što smo sortirali bili su elementi - ne nužno brojevi. Dakle, možemo zaključiti da algoritam za sortiranje ne ovisi o onome što se sortira. Doduše, jednim dijelom ovisi; naime, da bi sortiranje imalo smisla, moramo uvesti nekakvu relaciju uređenosti između elemenata; moramo na neki način definirati koji je element u kakvom odnosu sa ostalima. U svim prethodnim primjerima radili smo sa brojevima i relacijom veći od (ili manji od). Ispitivanje je li a manje od b jednostavno smo napisali if(a<b)... Da smo htjeli napraviti sort obrnutim redoslijedom, bilo bi potrebno prepraviti funkciju tako da sada umjesto if(a<b) ispituje if(a>b). Sve ostalo ostaje nepromijenjeno! Ovaj zaključak otvara mjesta izradi funkcija za sortiranje koje implementiraju određeni algoritam ali postižu i podatkovnu neovisnost! Drugim riječima, nije ih briga što sortiraju! Naravno, da bi ovo mogle, trebaju malu pomoć - netko im treba reći kako da usporede dva elementa polja! Za ovaj dio posla napraviti ćemo za tu priliku posebnu funkciju definiranu na slijedeći način:

prototip funkcije:

```
int usporedna_funkcija( const void *elem1, const void *elem2 );
```

Funkcija vraća:

- pozitivan broj, ukoliko je **elem1* > **elem2*
- nulu, ukoliko je **elem1* == **elem2*
- negativan broj, ukoliko je **elem1* < **elem2*

Oznaka > označava da se **elem1* mora pojaviti iza **elem2*, dok oznaka < označava da se **elem1* mora pojaviti ispred **elem2*.

Funkcija prima pokazivače na zadana dva elementa kojima treba odrediti odnos. Prima se pokazivač (void*) tipa zbog univerzalnosti - objasniti ću poslije. No pokažimo sada kako ćemo napraviti funkciju koja će usporediti dva integera, i to tako da značenje ">" iz definicije funkcije bude doista veće od, a "<" manje od. Evo koda:

```
int UsporediBrojevePrviManji(const void *elem1, const void *elem2) {
 return *((int*)elem1) - *((int*)elem2);
}
```

Kako naša funkcija prima pokazivače void* tipa, a znamo da oni pokazuju na integere, pokazivače ukalupljujemo u int* tip. Dalje, ako imamo pokazivač na neku vrijednost, tu vrijednost ćemo dobiti dereferenciranjem, dakle, *((int*)elem1) je broj koji se nalazi na adresi na koja je pohranjena u pokazivač elem1. Isto vrijedi i za *((int*)elem1). Kako po definiciji želimo vratiti pozitivnu vrijednost kada je prvi broj veći od drugoga, dovoljno će biti da ta dva broja oduzmemo! Tada izraz:

**((int*)elem1) - *((int*)elem2)*

ima pozitivnu vrijednost ako je broj pohranjen na adresi iz elem1 veći od broja pohranjenog na adresi iz elem2, 0 ako je broj pohranjen na adresi iz elem1 jednak broju pohranjenom na adresi iz elem2, a inače ima negativnu vrijednost!

Kako ćemo napraviti funkciju koja će brojeve uspoređivati, ali tako da > znači manje od? Pa jednostavno možemo prepisati gornji kod funkcije, i ispred izraza koji vraća rezultat staviti minus! Evo koda:

```
int UsporediBrojevePrviVeci(const void *elem1, const void *elem2) {
 return -(*((int*)elem1) - *((int*)elem2));
}
```

Kako bismo napravili funkciju koja će usporediti dva znakovna niza na klasičan način, tj. ako je prvi niz manji od drugog, vratiti negativan broj, ako su nizovi jednaki vratiti 0, a inače vratiti pozitivan broj? Pa možemo iskoristiti već postojeću funkciju strcmp! Ona radi baš taj posao! Evo koda:

```
int UsporediNizovePrviManji(const void *elem1, const void *elem2) {
 return strcmp((const char*)elem1, (const char*)elem2 );
}
```

a obratan poredak nizova daje:

```
int UsporediNizovePrviVeci(const void *elem1, const void *elem2) {
 return -strcmp((const char*)elem1, (const char*)elem2 );
}
```

Evo, sada smo vidjeli kako napravi funkcije koje odgovaraju zadanoj definiciji, a rade različite vrste usporedbi. Sada još treba napisati ono što sam najavio - podatkovno neovisnu funkciju za sortiranje. Evo ideje. Funkciji ćemo predati kao i uvijek pokazivač na polje te veličinu polja. No budući da funkcija mora premještati elemente polja, moramo predati i podatak koliko je pojedini element velik! I konačno, budući da sama sort funkcija ne zna uspoređivati elemente, prebamo joj reći koju funkciju treba zvati da bi se izvršila usporedba. Kao primjer algoritma za sortiranje poslužiti će nam onaj koji nas je doveo do BubbleSort1 funkcije. Deklaracija same funkcije izgleda malo zastrašujuće, ali ne dajte se smesti - sve je to trivijalno. Evo koda:

```
void BubbleSortGeneric( void *array, int elem_size, int n,
                        int (*f)(const void *elem1, const void *elem2) ) {
 int i,j,m;
 void *temp;

 temp = malloc( elem_size );
 if( temp == NULL ) return;
 for( i = 0; i < n-1; i++ ) {
  m = i;
  for( j = i+1; j < n; j++ ) {
   if( f((char*)array+m*elem_size,(char*)array+j*elem_size) > 0 ) m = j;
   //if( array[m] > array[j] ) { m = j; }
  }
  if( m != i ) {
   memcpy(temp,(char*)array+m*elem_size,elem_size);
   // temp = array[m];
   memcpy((char*)array+m*elem_size,(char*)array+i*elem_size,elem_size);
   // array[m] = array[i];
   memcpy((char*)array+i*elem_size,temp,elem_size);
   // array[i] = temp;
  }
 }
 free(temp);
}
```

Kod je identičan kodu funkcije *BubbleSort1*, osim dvije male izmjene; ispod mjesta koja su nanovo napisana ostavljen je komentiran stari kod tako da vidite što se zapravo događa u funkciji. Budući da je privremeni element nepoznatog tipa, ne može ga se deklarirati kao varijablu, već definiramo pokazivač, pa onoga trena kada se funkcija pozove, alocira se mjesto za jedan element jer se sada zna koliko je element velik - to govori parametar *elem_size*! Za usporedbu dva elementa zove se funkcija *f*, a zamjenu dva elementa obavljaju tri poziva funkcija *memcpy*. Naposlijetku, nakon što se sortiranje završi, oslobađa se alocirana memorija za pomoćni element. I možda da pojasnim još kako se dođe do adrese nekog od elemenata. Neka je *array* pokazivač na prvi bajt prvog elementa, i neka je svaki element velik *elem_size*. Na kojoj se adresi nalazi drugi element polja? Pa očito na adresi *array* uvečanoj za *elem_size* bajtova! No kako je *array void** tipa, da bismo dobili povećanje za neki broj bajtova, pokazivač najprije treba ukalupiti u *char** tip jer jedan znak zauzima jedan bajt, pa će ovo raditi ispravno (stvar je potrebna zbog C-ove aritmetike s pokazivačima). Općenito, *i*-ti element se nalazi na adresi *(char*)array+i*elem_size*.

I dobro, kako se ovo čudo koristi? Evo primjera:

```
void main( void ) {
  int polje[5];

  polje[0] = 7;
  polje[1] = 5;
  polje[2] = 2;
  polje[3] = 9;
  polje[4] = 8;
  BubbleSortGeneric( polje, sizeof(int), 5, UsporediBrojevePrviManji );
}
Rezultira sa:
polje[0] = 2;
polje[1] = 5;
polje[2] = 7;
polje[3] = 8;
polje[4] = 9;
```

Primjer 2.

```
void main( void ) {
  int polje[5];

  polje[0] = 7;
  polje[1] = 5;
  polje[2] = 2;
  polje[3] = 9;
  polje[4] = 8;
  BubbleSortGeneric( polje, sizeof(int), 5, UsporediBrojevePrviVeci );
}
Rezultira sa:
polje[0] = 9;
polje[1] = 8;
polje[2] = 7;
polje[3] = 5;
polje[4] = 2;
```

Primjer 3. Sortiranje nizova.

```
void main( void ) {
  char polje[POLJE_ELEM][10] = {
   "Peric",
   "Anic",
   "Ivic",
   "Kresic",
   "Maric"
  };

  BubbleSortGeneric( polje, 10, 5, UsporediNizovePrviManji );
}
Rezultira sa:
polje[0] = "Anic";
polje[1] = "Ivic";
polje[2] = "Kresic";
polje[3] = "Maric";
polje[4] = "Peric";
```

I sl. U standardnoj C biblioteci stdlib imate realiziranu podatkovno neovisnu implementaciju algoritma qsort, koji zahtjeva iste parametre kao i naša BubbleSortGeneric funkcija! Pogledajte!

QUICK SORT

Još sam Vam ostao dužan objasniti kako radi QuickSort. Pa evo ga. Osnovna ideja je slijedeća: rekurzivno primjenjivati slijedeći postupak: polje treba podijeliti na dva podpolja, na taj način da odaberemo jedan stožerni element, te polje uredimo tako da sve elemente koji su manji od stožernog stavimo ispred stožernog, a sve elemente koji su veći od stožernog stavimo iza stožernog. Na ovako nastala dva podpolja (od prvog elementa do stožernog, ali bez stožera, i od prvog elementa iza stožera do kraja) potrebno je ponovno primijeniti QuickSort rekurzivno. Možda je definicija malo smotana, ali to je ideja cijelog algoritma. Evo jednog primjera: treba sortirati polje {50, 21, 17, 16, 15, 13, 2}

Jednostavnosti radi, za stožer ćemo uzimati onaj broj koji se zatekne na sredini polja. Zatim ćemo provjeriti vrijedi li da je prvi element polja manji od stožera i manji od zadnjeg elementa polja, a stožer manji od zadnjeg elementa polja. Ako nešto od ovoga ne vrijedi, jednostavno ćemo zamijeniti elemente tako da osiguramo da dana pretpostavka vrijedi. Pa krenimo redom:

1. Korak
Polje: **50**, 21, 17, **16**, 15, 13, **2**
Srednji član je 16; biramo ga za stožer.
Provjeravamo: 50<16 ==> nije, zamjenjujemo ih.
Polje: **16**, 21, 17, **50**, 15, 13, **2**
Provjeravamo 16<2 ==> nije, zamjenjujemo ih.
Polje: **2**, 21, 17, **50**, 15, 13, **16**
Provjeravamo 50<16 ==> nije, zamjenjujemo ih.
Polje: **2**, 21, 17, **16**, 15, 13, **50**
Ovime smo odabrali kao stožer broj 16. Sada moramo sve elemente koji su manji od njega staviti na lijevu stranu a sve veće na desnu. Kako ćemo ovo izvesti, ovisiti će o implementaciji same funkcije, ali za sada odaberimo proizvoljno:
Polje: **2**, 15, 13, **16**, 21, 17, **50**
Ovime smo dobili dva podpolja: {2,15,13} i {21,17,50} koja su razdvojena stožerom 16. Sada na svako polje treba primijeniti ponovno QuickSort algoritam!

2-1.Korak - lijevo podpolje.
Polje: **2**, **15**, **13**, 16, 21, 17, 50
Srednji član je 15; biramo ga za stožer.
Provjeravamo: 2<15 ==> je.
Provjeravamo 2<13 ==> je.
Provjeravamo 15<13 ==> nije, zamjenjujemo ih.
Polje: **2**, **13**, **15**, 16, 21, 17, 50
Ovime smo odabrali kao stožer broj 13. Sada moramo sve elemente koji su manji od njega staviti na lijevu stranu a sve veće na desnu. No budući da imamo samo tri elementa, poredak očito vrijedi.
Ovime smo opet dobili dva podpolja: {2} i {15} koja su razdvojena stožerom 13. Sada na svako polje treba primijeniti ponovno QuickSort algoritam, no budući da polja imaju samo jedan element, postupak se zaustavlja. Ovime je sortiranje lijevoga podpolja gotovo. Sada još treba sortirati desno podpolje:

2-2.Korak - desno podpolje.
Polje: 2, 15, 13, 16, **21, 17, 50**
Srednji član je 17; biramo ga za stožer.
Provjeravamo: 21<17 ==> nije, zamjenjujemo ih.
Polje: 2, 15, 13, 16, **17, 21, 50**
Provjeravamo 17<50 ==> je.
Provjeravamo 21<50 ==> je.
Ovime smo odabrali kao stožer broj 21. Sada moramo sve elemente koji su manji od njega staviti na lijevu stranu a sve veće na desnu. No budući da imamo samo tri elementa, poredak očito vrijedi.
Ovime smo opet dobili dva podpolja: {17} i {50} koja su razdvojena stožerom 21. Sada na svako polje treba primijeniti ponovno QuickSort algoritam, no budući da polja imaju samo jedan element, postupak se zaustavlja. Ovime je sortiranje desnoga podpolja gotovo, i cijeli postupak sortiranja polja je gotov.

Pozabavimo se sada malo sa problemom kako ostvariti da se svi elementi koji su manji od stožera nađu lijevo, a oni veći desno. U našem primjeru baš je ispalo da stožer ostaje na svom mjestu - u sredini polja, no to općenito ne mora biti istina. Npr. uz polje {2,8,7,6,20,2,5,4,30} stožer bi bio 20 (na pola polja), a nakon uređivanja polja dobili bismo {2,8,7,6,2,5,4,20,30}! Stožer bi otputovao skroz na predzadnje mjesto! Jedna od mogućnosti rješavanja problema je slijedeća:

1. Osigurati da vrijedi:
   - prvi element je manji od stožera
   - prvi element je manji od zadnjega
   - stožer je manji od zadnjeg elementa

   Ako nešto od ovoga ne vrijedi, zamijeniti dotične elemente (čime će odnos vrijediti).

2. Zamijeniti stožer sa predzadnjim elementom polja.
3. Krenuti kroz polje od početka i tražiti element koji je veći od stožera (dakle koji remeti odnos da su lijevo elementi manji od stožera); kada se nađe element, preći na korak 4.
4. Krenuti kroz polje s kraja i tražiti element koji je manji od stožera (dakle koji remeti odnos da su desno elementi veći od stožera); kada se nađe element, preći na korak 5.
5. Ukoliko je pronađeni element u koraku 3 iza pronađenog elementa u koraku 4, tada je potrebno zamijeniti predzadnji element polja (stožer) sa elementom pronađenim u koraku 3; u suprotnom, treba zamijeniti elemente pronađene u koraku 3 i 4, i ponovno krenuti od koraka 3.

Opisani algoritam osigurati će nam dosta efikasno uređenje polja po zadanom kriteriju. Pokažimo sada konačno rad algoritma na nekom primjeru: {12, 21, 11, 30, 15, 17, 16}

1. polje: 12, 21, 11, 30, 15, 17, 16; biramo stožer na sredini polja ==> 30.
   - 12<30 ==> je.
   - 12<16 ==> je.
   - 30<16 == nije, radimo zamjenu! Izgled polja: 12, 21, 11, 16, 15, 17, 30
2. Zamjenjujemo stožer (16) sa predzadnjim elementom (17). Izgled polja: 12, 21, 11, 17, 15, 16, 30
3. Pronađen je element 21 koji je veći od stožera.
4. Pronađen je element 15 koji je manji od stožera.
5. Budući da je 21 ispred broja 15, zamjenjujemo ih. Izgled polja: 12, 15, 11, 17, 21, 16, 30. Vraćamo se ponovno na korak 3:

3. Pronađen je element 17 koji je veći od stožera.
4. Pronađen je element 11 koji je manji od stožera.
5. Budući da je 17 iza broja 11, zamjenjujemo broj 17 sa stožerom. Izgled polja: 12, 15, 11, 16, 21, 17, 30. Ovime se uređivanje polja prekida.

Sada kada smo vidjeli kako algoritam radi, idemo ga samo mrvicu poboljšati. Pogledajmo korak 3. Kaže se, "kreni od početka... " No je li potrebno krenuti od početka? Pa korak 1 nam osigurava da prvi element nije veći od stožera! Znači da možemo krenuti od drugog elementa! Dalje, ako u koraku 4 nađemo broj manji od stožera, a u koraku 5 napravimo zamjenu sa elementom pronađenim u koraku 3, tada ponovnu pretragu za korak 3 možemo započeti od prvog elementa koji je iza onog pronađenog u prethodnom koraku 3 (naime, svi prije njega bili su manji, a on je remetio poredak; no on je zamijenjen sa elementom pronađenim u koraku 4 pa je i on sada manji! Traženje se dakle može nastaviti od prvog elementa iza njega)! Slična priča vrijedi i za pretrage u koraku 4. Budući da nam korak 1 osigurava da je zadnji element veći od stožera, tada se pretraga može započeti od predzadnjeg elementa! Dodatno, korak dva na predzadnje mjesto stavi baš stožer pa i taj element očito nije manji od stožera! Tada i predzadnji element možemo preskočiti prilikom traženja, i stvarnu pretragu započeti od predpredzadnjeg elementa. Dalje, nakon što u koraku 4 pronađemo broji koji je manji od stožera, a u koraku 5 ga zamijenimo sa onim pronađenim u koraku 3, pretraga se ne mora ponovno startati od kraja polja, već od elementa koji je ispred pronađenog u koraku 4 (jer je u koraku 5 na njegovo mjesto stigao broj veći od stožera, a svi brojevi od

kraja pa do tog broja su već ionako veći jer smo to i ustanovili pretragom - dakle, nema je smisla ponavljati).

I konačno, programska implementacija algoritama slijedi. Kako će nam često trebati funkcija koja zamjenjuje dva elementa, napisati ćemo si malu pomoćnu funkciju za taj posao:

```
inline void Zamijeni(int *lijevo, int *desno ) {
 int pom = *lijevo;
 *lijevo = *desno;
 *desno = pom;
}
```

Ključna riječ inline govori kompajleru da to ne kompajlira kao klasičnu funkciju, već svaki poziv iste zamijeni na licu mjesta kodom funkcije. To Vam dođe isto kao da i niste pisali funkciju, već svaki puta napisali kompletan niz naredbi koji mi utrpamo u inline funkciju. Da li se ovakve funkcije dopuštaju ili ne, ovisi o kompajleru. Tako npr. Borland C++ 3.1 Vam ovo ne dopušta (dopušta samo za članske funkcije klasa), C++ Builder v1.0 dopušta, a u skripti iz ASP-a sam otkrio da umjesto ključne riječi inline koriste __inline (vjerojatno neka Microsoftova izmišljotina, no nisam siguran pa pitajte autora skripte gdje se to koristi).

Funkcija koja obavlja rekurzivni QuickSort prima tri argumenta: pokazivač na polje (ili točnije, na prvi bajt prvog elementa polja), početak podpolja (indeks prvog elementa koji pripada podpolju koje treba sortirati) te kraj podpolja (indeks zadnjeg elementa koji pripada podpolju koje treba sortirati). A naziv funkcije ne ističe moju egocentričnost (ili što bi već zlobne duše mogle pripisati), već naprosto činjenicu da ću Vam nakon ovoga dati i rješenje algoritma kakvo je predstavljeno u skripti iz ASP-a, pa da ne dođe do konfuzije, svaka funkcija ima svoje ime. Evo funkcije:

```
void MojQSort( int *array, int  left, int right ) {
 int i,j,middle,stozer;

 // Ako imam bar cetiri elementa polja
 if( left+3<= right ) {
  // izracunaj sredinu (poziciju na kojoj je stozer)
  middle = (left+right)/2;
  // ako je prvi element veci od stozera, zamijeni ih
  if(array[left]>array[middle]) Zamijeni(&array[left],&array[middle]);
  // ako je prvi element veci od zadnjega, zamijeni ih
  if(array[left]>array[right]) Zamijeni(&array[left],&array[right]);
  // ako je stozer veci od zadnjeg elementa, zamijeni ih
  if(array[middle]>array[right]) Zamijeni(&array[middle],&array[right]);
  // Sada kada smo osigurali poredak, znamo tko je pravi stozer
  stozer = array[middle];
  // Stavi stozer na predzadnje mjesto
  Zamijeni(&array[middle],&array[right-1]);
  // Idemo urediti polje; lijevo pocinjemo traziti od drugog elementa,
  // desno od predpredzadnjeg
  i=left+1; j=right-2;
  while(1) {
   // tako dugo dok je lijevo element manji od stozera, idemo dalje
   while( array[i]<stozer ) i++;
   // tako dugo dok je desno element veci od stozera, idemo dalje
   while( array[j]>stozer ) j--;
   // ako je manji element iza vecega, zamjeni ih i nastavi trazenje
   if( j>i ) Zamijeni(&array[i],&array[j]);
   // inace prekidamo trazenje
   else break;
  }
  // vrati stozer na svoju poziciju
  Zamijeni(&array[i],&array[right-1]);
  // i pozovi QuickSort nad novim podpoljima
  // :polje od pocetka pa do stozera ne ukljucujuci stozer
```

```
  MojQSort(array, left, i-1);
  // :polje od iza stozera pa do kraja
  MojQSort(array, i+1, right);
 } else {
  // Ukoliko imamo tri ili manje elemenata, umjesto QuickSorta
  // brzo cemo obaviti i klasicno sortiranje
  UbaciSort(array+left,right-left+1);
 }
}

// Funkcija koja poziva rekurzivnu implementaciju algoritma
void MojQuickSort( int *array, int n ) {
 MojQSort( array, 0, n-1 );
}
```

Nadam se da je sve jasno! Što se rješenja u skripti tiče, ono je malo "razbijenije" od ovoga. Funkcija *medijan3* implementira Vam točke 1 i 2 opisanih pravila (na početku teksta, 5 točaka). Zatim funkcija *QSort* implementira ostatak pravila i brine za rekurzivno pozivanje funkcije. I konačno funkcija *QuickSort* predstavlja most između Vas i rekurzivne verzije iste, kao što je i kod mene postojala *MojQuickSort*. Evo funkcija:

```
int medijan3( int *array, int left, int right ) {
 int middle = (left+right)/2;

 if(array[left]>array[middle]) Zamijeni(&array[left],&array[middle]);
 if(array[left]>array[right]) Zamijeni(&array[left],&array[right]);
 if(array[middle]>array[right]) Zamijeni(&array[middle],&array[right]);
 Zamijeni(&array[middle],&array[right-1]);
 return array[right-1];
}

void QSort( int *array, int left, int right ) {
 int i,j,stozer;

 if( left+3<=right ) {
  stozer = medijan3(array, left, right );
  i=left; j=right-1;
  while(1) {
   while( array[++i] < stozer );
   while( array[--j] > stozer );
   if( i<j ) Zamijeni(&array[i],&array[j]);
   else break;
  }
  Zamijeni(&array[i],&array[right-1]);
  QSort(array,left,i-1);
  QSort(array,i+1,right);
 } else {
  UbaciSort(array+left,right-left+1);
 }
}

void QuickSort( int *array, int n ) {
 QSort( array, 0, n-1 );
}
```

Za kod iz skripte dao sam si malo truda da biste mogli pratiti što se doista događa sa elementima polja, pa sam dodao par kontrolnih mjesta i različitih ispisa. Traženo je sortiranje polja brojeva {50, 21, 17, 16, 15, 13, 2}. Kako ovo što slijedi predstavlja sliku izvođenja programa, polja su indeksirana od 0 - nemojte to zaboraviti. Evo što se je događalo:

```
Pozvan QSort sa granicama 0 do 6
Izgled polja: 50, 21, 17, 16, 15, 13,  2
Trazim stozer...
Pocetak trazenja stozera ***************
 Izgled polja: 50, 21, 17, 16, 15, 13,  2
 Kandidat za stozer je 16
 Zamjenjujem elemente polja 50 i 16
 Izgled polja: 16, 21, 17, 50, 15, 13,  2
 Novi kandidat za stozer je 50
 Zamjenjujem elemente polja 16 i 2
 Izgled polja:  2, 21, 17, 50, 15, 13, 16
 Zamjenjujem elemente polja 50 i 16
 Izgled polja:  2, 21, 17, 16, 15, 13, 50
 Novi kandidat za stozer je 16
 Guram stozer 16 na predzadnje mjesto.
 Izgled polja:  2, 21, 17, 13, 15, 16, 50
Kraj trazenja stozera ***************
nasao. Izgled polja:  2, 21, 17, 13, 15, 16, 50
Stozer = 16
Zamjenjujem 21 sa 15
Izgled polja:  2, 15, 17, 13, 21, 16, 50
Zamjenjujem 17 sa 13
Izgled polja:  2, 15, 13, 17, 21, 16, 50
Vracam stozer 16. Izgled polja:  2, 15, 13, 16, 21, 17, 50
Pozvan QSort sa granicama 0 do 2
Zovem UbaciSort. Rezultat: Izgled polja:  2, 13, 15, 16, 21, 17, 50
Pozvan QSort sa granicama 4 do 6
Zovem UbaciSort. Rezultat: Izgled polja:  2, 13, 15, 16, 17, 21, 50
```

Ukoliko polju dodamo još koji element, moći ćemo promatrati pravu rekurziju QSort funkcije! Uzmimo novo polje {50, 21, 17, 16, 15, 13, 2, 30, 1}. Evo ispisa:

```
Pozvan QSort sa granicama 0 do 8
Izgled polja: 50, 21, 17, 16, 15, 13,  2, 30,  1
Trazim stozer...
Pocetak trazenja stozera ***************
 Izgled polja: 50, 21, 17, 16, 15, 13,  2, 30,  1
 Kandidat za stozer je 15
 Zamjenjujem elemente polja 50 i 15
 Izgled polja: 15, 21, 17, 16, 50, 13,  2, 30,  1
 Novi kandidat za stozer je 50
 Zamjenjujem elemente polja 15 i 1
 Izgled polja:  1, 21, 17, 16, 50, 13,  2, 30, 15
 Zamjenjujem elemente polja 50 i 15
 Izgled polja:  1, 21, 17, 16, 15, 13,  2, 30, 50
 Novi kandidat za stozer je 15
 Guram stozer 15 na predzadnje mjesto.
 Izgled polja:  1, 21, 17, 16, 30, 13,  2, 15, 50
Kraj trazenja stozera ***************
nasao. Izgled polja:  1, 21, 17, 16, 30, 13,  2, 15, 50
Stozer = 15
Zamjenjujem 21 sa 2
Izgled polja:  1,  2, 17, 16, 30, 13, 21, 15, 50
Zamjenjujem 17 sa 13
Izgled polja:  1,  2, 13, 16, 30, 17, 21, 15, 50
Vracam stozer 15. Izgled polja:  1,  2, 13, 15, 30, 17, 21, 16, 50
Pozvan QSort sa granicama 0 do 2
```

```
 Zovem UbaciSort. Rezultat: Izgled polja:  1,  2, 13, 15, 30, 17, 21, 16, 50
Pozvan QSort sa granicama 4 do 8
 Izgled polja:  1,  2, 13, 15, 30, 17, 21, 16, 50
 Trazim stozer...
 Pocetak trazenja stozera ***************
  Izgled polja:  1,  2, 13, 15, 30, 17, 21, 16, 50
  Kandidat za stozer je 21
  Zamjenjujem elemente polja 30 i 21
  Izgled polja:  1,  2, 13, 15, 21, 17, 30, 16, 50
  Novi kandidat za stozer je 30
  Guram stozer 30 na predzadnje mjesto.
  Izgled polja:  1,  2, 13, 15, 21, 17, 16, 30, 50
 Kraj trazenja stozera ***************
 nasao. Izgled polja:  1,  2, 13, 15, 21, 17, 16, 30, 50
 Stozer = 30
 Vracam stozer 30. Izgled polja:  1,  2, 13, 15, 21, 17, 16, 30, 50
 Pozvan QSort sa granicama 4 do 6
  Zovem UbaciSort. Rezultat: Izgled polja:  1,  2, 13, 15, 16, 17, 21, 30, 50
 Pozvan QSort sa granicama 8 do 8
  Zovem UbaciSort. Rezultat: Izgled polja:  1,  2, 13, 15, 16, 17, 21, 30, 50
```

KAKO RIJEŠAVATI ISPITE VEZANE UZ C

Potaknut brojnim pitanjima a i vlastitim primjedbama odlučio sam riješiti jedan ispit iz Algoritama i struktura podataka, te uputiti na klasične principe i načine rješavanja pojedinih zadataka. Ovaj tekst namijenjen je prvenstveno studentima, no kako u tekstu govorim i o nekim nedostacima samih ispita, bilo bi poželjno da na tekst oko bace i neke od odgovornih osoba. Ispit koji ću riješiti je sa roka 1. veljače 1999. Dodatno, ovaj tekst nije ovdje zato da bi rekao kako sastavljači ispita nemaju blage veze o tome što rade, i svako takvo tumačenje je krivo! Tekst se nalazi ovdje prvenstveno zato da bi upozorio na stvari koje se događaju, a ne bi smjele. Preporuke studentima nalaze se u plavim okvirima, dok se zapažanja za sastavljače nalaze u crvenim okvirima.

ZADATAK 1.

Napisati funkcije za upis u strukturu red i uzimanje podataka iz te strukture. Podatak neka bude znakovni niz. Red se realizira kao jednostruko povezana linearna lista.

NAPUTAK:
Novi zapis dodaje se na kraj liste. Ulazni podaci za funkciju dodavanja su pokazivač na sadržaj novog zapisa i pokazivač na poslijednji zapis u listi. Funkcija za brisanje ima kao ulazni podatak glavu liste. Vraća sadržaj prvog zapisa u listi te ga briše iz liste.

Pogledajmo prvo što se u zadatku traži. Zadana je struktura **RED**. To znači radimo sa strukturom koja je FIFO tipa (sjećate se, First In, First Out). Na predavanjima se red radi u statičnom polju, dok je ovdje zadana lista, no to nas ne smije preplašiti. Sjetimo se kako se radi sa listama. Obično imamo pokazivač na prvi element liste, a dodavanja novih čvorova liste mogu ići bilo gdje. No kako bismo morali raditi sa listom da dobijemo efekt *reda*? Pa jedna od očitih ideja je slijedeća: u listu ćemo dodavati na kraj, a elemente ćemo vaditi od početka. Doista, ako u listu dodamo element A, pa iza njega element B, pa iza element C, itd. dobiti ćemo listu oblika A-B-C-... Krenimo sada čitati elemente ali od početka. Čitamo A, ostane B-C-..., čitamo B, ostane C-..., itd. Dakle elemente čitamo upravo onim redom kako smo ih upisivali. To je dakle FIFO struktura, odnosno RED. Sada kada smo shvatili o čemu se radi, krenimo na prvu funkciju.

Trebamo realizirati upis podatka u strukturu, i kao parametre dobivamo podatak i pokazivač na trenutno zadnji čvor liste. Pa po gornjoj ideji to je sve što nam i treba! Naime, kako zadnji čvor više neće biti zadnji, sve što trebamo učiniti jest *Next* član tog čvora postaviti tako da pokazuje na član koji dodajemo, a čvoru koji dodajemo treba član *Next* postaviti na NULL jer on doista jest zadnji. Pa krenimo na posao. Prvo moramo napisati protip funkcije. Funkcija ne vraća ništa. Kao ulazni podatak proslijeđuje se pokazivač na poslijednji zapis, i pokazivač na *novi sadržaj*. Iz ovog teksta zaključujemo da prototip ne možemo napisati! Naime, što je to pokazivač na novi sadržaj? Sadržaj koji pamtimo jest znakovni niz, pa je to valjda pokazivač na prvi znak niza koji želimo pohraniti! Ili ipak nije? Pokazivač na novi sadržaj može se isto tako protumačiti kao pokazivač na novo alocirani čvor u koji je već upisan željeni znakovni niz! Za koji ćemo se opciju odlučiti? Nadalje, prije nego što se uopće odlučimo govoriti o čvorovima, potrebno je napisati strukturu tog čvora. Pa krenimo redom. Prvo imamo pokazivač Next na slijedeći element strukture. Zatim tu dolazi jedan znakovni niz, dakle polje znakova veliko ? Ups! Niti ovo ne znamo! No nije važno. Možda da samo stavimo kao ovaj element jedan pokazivač na niz, a taj pokazivač ćemo inicijalizirati tako da pokazuje na predani niz, pa nam nije uopće važno koliko je niz velik, jer se je za alokaciju odgovarajućeg prostora već pobrinuo pozivatelj (ili ipak nije?). A što ako pozivatelj zapise čita iz npr. neke datoteke? Uobičajeno riješenje je deklarirati znakovno polje od npr. 1024 znakova i u njega čitati red po red, te pozivati funkciju dodaj sa pokazivačem na to polje. No tada nam opet implementacija ne valja jer mi pamtimo samo pokazivač na niz, a on se svakim čitanjem mijenja! Tada bi moguće riješenje bilo svaki puta kada se pozove funkcija dodaj, u funkciji alocirati novu kopiju predanog niza i pamtiti pokazivač na njega! Tada ako se original i promjeni, to neće utjecati na naš red jer smo mi zapamtili original onakav kakav je on bio u trenutku kada je pozvana funkcija dodaj.

Zadatak uvijek treba biti zadan točno i precizno. Ako je student u mogučnosti da razmišlja je li sastavljač mislio ovo ili ono, tada takav zadatak nije dobro zadan! Konkretno na ovom primjeru, pitanja:

- je li pokazivač na novi sadržaj (liste) pokazivač na prvi znak znakovnog niza, ili pokazivač na gotovu strukturu (čvor) u koju je već pohranjen zadani niz?

ako je predan pokazivač na prvi znak zadanog niza

- da li je taj niz jedinstveno alociran za naš red i kao takav se neće mijenjati?
- ili je taj niz pokazivač na neki buffer koji će se mijenjati a mi moramo stvoriti i zapamtiti novu kopiju niza?

uopće nisu glupa pitanja! I dapače, svaki malo iskusniji programer će automatski uvidjeti ova pitanja i dalje neće znati što da napravi! Uočite da ovo nije neznanje onoga koji riješava ispit već vaš propust da jasno i jednoznačno zadate što želite od programera.

Kako smo ovdje naveli puno previše mogućih pretpostavki, očito je da ne možemo dati jedinstvenu implementaciju zadane funkcije. Zato ćemo napisati dvije moguće implementacije i uz svaku navesti uvijete i pretpostavke pod kojima smo funkciju napisali. Od sada na dalje odustajemo od jedne mogućnosti: struktura neće pamtiti zadani niz preko nekog svog znakovnog polja. Od ovoga moramo odustati jer ne možemo znati koliko znakova maksimalno može zadani niz znakova imati. Opcija koja nam je ostala jest pamtiti običan pokazivač na niz. No i tu imamo dalje nekoliko mogučnosti.

PRETPOSTAVKA PRVA.
Pamtiti ćemo zadani niz znakova preko pokazivača na isti. Dodatno, funkciji se predaje jedinstven niz znakova koji se u programu neće mijenjati. Pod pokazivač na sadržaj smatrati ćemo pokazivač na prvi znak znakovnog niza koji trebamo upamtiti.

```
typedef struct CvorStr {
  struct CvorStr *Next;
  char *Niz;
} Cvor;

void Dodaj( char *Niz, Cvor *Zadnji ) {
 Cvor *cv;

  cv = (Cvor*)malloc(sizeof(Cvor));
  cv->Next = NULL;
  cv->Niz = Niz;
  Zadnji->Next = cv;
}
```

Eto, malo koda i to je to. Ili ipak? Nagradno pitanje broj jedan: što će se dogoditi ako je red prazan, pa zadnji element uopće ne postoji? Tada je pokazivač Zadnji postavljen na NULL pa nema smisla reći Zadnji->Next=cv!!! No dobro, riješenje glasi,
if(Zadnji != NULL) Zadnji->Next = cv;
Sada malo zastanite i razmislite. Može li se dogoditi da je red prazan? Pa dakako da može! To je jedna zrela i točno pretpostavka. Idemo korak dalje. Ako dodajemo element u red koji je prazan, očito je da će dodani element biti zadnji! No mi nemamo načina da promijenimo pokazivač zadnji i napišemo ono što bi bilo ispravno, jer neznamo koja je adresa pokazivača Zadnji da bismo ovu izmjenu mogli implementirati!

Ako navodite koje parametre treba primiti pojedina funkcija, obratite pažnju na to da zadate takve parametre uz koje će se funkcija uopće moći realizirati. Tekst zadatka kaže: funkcija prima pokazivač na poslijednji zapis u listi. Uz ovakav parametar dotična se funkcija jednostavno ne može realizirati ispravno! Ono što funkcija mora primiti jest adresu pokazivača na zadnji član da bi taj zadnji član mogla

promijeniti! No oni studenti koji i dođu do ovog zaključka, neće se usuditi ovo napisati jer ste parametre zadali Vi, i to krivo.

Sada slijedi nagradno pitanje broj 2: što će se još dogoditi ako je red prazan a mi dodajemo novi element, osim onoga opisanoga pod odgovorom na prvo pitanje? Pa pogledajmo stvar malo iz druge perspektive. Ako u prazan red dodajemo novi element, tada je taj element ujedno i zadnji element liste i reda. No kao što rekoh, pogledajmo stvar iz malo druge perspektive. Ako je lista ima samo jedan element, i on je zadnji, koju biste mu medalju dodijelili za to mijesto? Na glupog li pitanja. ZLATO! Naravno. Zato što je taj element ujedno i prvi! No sada smo opet u dilemi. Ako je lista bila prazna, tada je i pokazivač Prvi bio jednak NULL, no nakon dodavanja u listi postoji jedan element, te Prvi treba postaviti da pokazije na taj element! No kako da to učinimo kada naša funkcija čak niti ne zna da pokazivač Prvi postoji? Ja ga ne vidim nigdje u parametrima funkcije!

Prije nego što zaključite listu parametara funkcije, provjerite da li se funkcija može realizirati uz zadane parametre. Na ovo nećete izgubiti puno vremena, a dobit je višestruka: zadali ste ispravan zadatak, i postoji mogučnost da ga netko zapravo i riješi ispravno.

Kako bi dakle trebao izgledati prototip zadane funkciije i njeno ispravno riješenje?

```
void Dodaj( char *Niz, Cvor **Prvi, Cvor **Zadnji ) {
 Cvor *cv;

  cv = (Cvor*)malloc(sizeof(Cvor));
  cv->Next = NULL;
  cv->Niz = Niz;
  if( *Zadnji == NULL ) {  // tada je i *Prvi == NULL
   *Prvi = cv;
   *Zadnji = cv;
  } else {
   (*Zadnji)->Next = cv;
   // Slijedeci redak se može dodati ako nam je dopušteno UVIJEK dirati
pokazivač Zadnji
   *Zadnji = cv;
  }
}
```

Uz iste pretpostavke i uvijete kao i za funkciju Dodaj, napišimo sada funkciju Vrati. Kako je u tekstu zadatka eksplicitno rečeno da funkcija prima glavu liste, a po definiciji glava liste je pokazivač na prvi član liste, vidjeti ćemo da se niti ova funkcija ne može realizirati uz zadane parametre.

```
char *Vrati( Cvor *Prvi ) {
  char *Niz;

  if( Prvi == NULL ) return NULL; // greska, lista je već prazna! Inaće,
  Niz = Prvi->Niz;
  free( Prvi );
  return Niz;
}
```

Zašto ovo ne valja? Pa za početak, ako izvadimo prvi čvor liste, tada on više nije prvi nego je novi prvi čvor liste zapravo Prvi->Next. No mo nemamo načina da ovo ikome javimo. Dodatno, ako lista ima samo jedan čvor, njegovim brisanjem lista postaje prazna, pa bi pokazivač Prvi očito trebalo postaviti na NULL, ali isto tako bi i pokazivač Zadnji trebalo postaviti na NULL jer u praznoj listi nema zadnjeg čvora! No niti ovo nismo u mogučnosti napraviti jer funkcija Vrati čak niti ne prima pokazivač Zadnji, a kamoli tek njegovu adresu! Dakle, ispravno riješenje ove funkcije glasi:

```
char *Vrati( Cvor **Prvi, Cvor **Zadnji ) {
  char *Niz;
  Cvor *cv;

  if( *Prvi == NULL ) return NULL; // greska, lista je već prazna! Inaće,
  Niz = (*Prvi)->Niz;
  cv = (*Prvi)->Next;
  free( *Prvi );
  *Prvi = cv;
  if( cv == NULL ) *Zadnji = NULL; // a tada je i *Prvi == NULL zbog linije
ispred, i cv == NULL!
  return Niz;
}
```

Ne koristite memoriju kojoj niste vlasnici! Naime, nekima će se činiti da je pokazivač cv običan višak koji dodatno opterećuje kod. Ti isti bi dotičnu funkciju napisali ovako:

```
char *Vrati( Cvor **Prvi, Cvor **Zadnji ) {
  char *Niz;

  if( *Prvi == NULL ) return NULL; // greska, lista je već prazna! Inaće,
  Niz = (*Prvi)->Niz;
  free( *Prvi );
  *Prvi = (*Prvi)->Next;
  if( *Prvi == NULL ) *Zadnji = NULL;
  return Niz;
}
```

Ovaj kod počiva na pretpostavci da operativni sustav neće stići upisati ništa novoga u memoriju koju ste upravo oslobodili, barem još neko vrijeme. Koristeći tu činenicu, prvo oslobađate strukturu na koju pokazuje *Prvi pozivom free(*Prvi), a nakon toga ipak pristupate toj strukturi i komadu memorije gdje se ona nalazi pretpostavljajuči kako operativni sustav još nije stigao ništa izmjeniti, pa kažete *Prvi = (*Prvi)->Next. I ovo je ono gdje čovjek griješi. Istina jest da se sadržaj tek oslobođene memorije obično ne mijenja odmah. No oslanjajući se na obično, čovijek obično i loše prođe! Dobro zapamtite gornje pravilo! **Ne koristite memoriju kojoj niste vlasnici!**

PRETPOSTAVKA DRUGA.
Pamtiti ćemo zadani niz znakova preko pokazivača na isti. Dodatno, funkciji se **ne** predaje jedinstven niz znakova, pa je potrebno pamtiti kopiju niza. Pod pokazivač na sadržaj smatrati ćemo pokazivač na prvi znak znakovnog niza koji trebamo upamtiti.

Prikazati ćemo odmah kompletno riješenje sa obje funkcije, uz uvažavanje svih primjedbi do kojih smo došli kroz prvu pretpostavku (uključujući i ispravke prototipova funkcija).

```
typedef struct CvorStr {
  struct CvorStr *Next;
  char *Niz;
} Cvor;

void Dodaj( char *Niz, Cvor **Prvi, Cvor **Zadnji ) {
 Cvor *cv;
 char *NizCpy;
 size_t l;

  l = strlen(Niz);
  NizCpy = (char*)malloc(l+1);
  if( NizCpy == NULL ) return;
  strcpy( NizCpy, Niz );
```

```
  cv = (Cvor*)malloc(sizeof(Cvor));
  if( cv == NULL ) { free(NizCpy); return; }
  cv->Next = NULL;
  cv->Niz = NizCpy;
  if( *Zadnji == NULL ) {  // tada je i *Prvi == NULL
   *Prvi = cv;
   *Zadnji = cv;
  } else {
   (*Zadnji)->Next = cv;
   // Slijedeci redak se može dodati ako nam je dopušteno UVIJEK dirati
pokazivač Zadnji
   *Zadnji = cv;
  }
}

char *Vrati( Cvor **Prvi, Cvor **Zadnji ) {
  char *Niz;
  Cvor *cv;

  if( *Prvi == NULL ) return NULL; // greska, lista je već prazna! Inaće,
  Niz = (*Prvi)->Niz;
  cv = (*Prvi)->Next;
  free( *Prvi );
  *Prvi = cv;
  if( cv == NULL ) *Zadnji = NULL; // a tada je i *Prvi == NULL zbog linije
ispred, i cv == NULL!
  return Niz;
}
```

Funkcija Dodaj pretrpjela je neke izmjene utoliko što smo dodali kod koji alocira novu kopiju zadanog niza i nju pamti a ne originalan tekst. Funkcija Vrati ostala je identična. Kao što vidite, ona nigdje ne oslobađa kopiju koja je alocirana u funkciji Dodaj, već tu kopiju vraća pozivatelju funkcije koji je dalje odgovoran da nakon što završi sa uporabom, kopiju oslobodi pomoću funkcije free. Razlog ovome je očit! Ne možemo najprije osloboditi kopiju iz memorije, a onda vratiti pozivatelju pokazivač na oslobođenu memoriju!

Kada naiđete na zadatak gdje morate pretpostavljati, obavezno napišite koje su sve pretpostavke moguće, te koje ste pretpostavke Vi odlučili usvojiti. Naime, osim ako niste telepat, ne možete znati kako je zadatak sebi zamislio onaj tko ga je (očito loše) sastavio. Ako vi samo napišete jedno, a sastavljač si je zamislio drugo, pozdravite se od bodova! Osim ako... **Zaštitite sebe**! Pripremite si dovoljno materijala da možete pobiti svaki prigovor onoga tko ocijenjuje zadatke. Ako je on zamislio jedno, pokažite mu da ste se vi sjetili i toga, ali kako niste mogli znati što si je tko zamislio, odabrali ste jedan od načina. Dobro dokumentirajte sve moguće pretpostavke, i tada odaberite jednu, te još jednom napišite: program je realiziran uz pretpostavke xyz.

Dodatno, jednog lijepog sunčanog dana kada počnete sami pisati nekakve programe i funkcije u C-u, sjetite se samo kako je ovdje realizirana funkcija Dodaj, i onda ponavljajte za mnom: ja neću tako pisati funkcije, ja neću tako pisati funkcije, ja neću tako pisati funkcije.... Funkcija dodaj ima jedan nedostatak koji si jednostavno ne možete dozvoliti: nema nikakve povratne vrijednosti kojom bi indicirao uspješnost obavljanja zadanog posla! Naša funkcija radi sa memorijom. Štoviše, ona ALOCIRA memoriju! Pa ovo je jedno od mjesta koje uvijek može ne uspjeti! I što vi radite? Kao nojevi zabijete glavu u pijesak da ne bi vidjeli što se je dogodilo, i jednostavno nastavite sa radom programa kao da se ništa nije dogodilo! Osim što naravno u redu nema zadanih nizova, doista se ništa drugo i nije dogodilo. A naravno, program koji ne radi ono za što je napisan možete slobodno:

- pohraniti na diskete u dvije kopije
- napraviti backupe na zip-drive
- jednu kopiju ispržiti na CD

- sve skupa uzeti u jednu ruku i baciti u koš za smeće

Eto, toliko o riješenju prvog zadatka.

ZADATAK 2.

Napisti rekurzivnu funkciju za izračunavanje n-tog Fibonaccijevog broja i navesti približnu apriornu složenost algoritma.

Fibonaccijevi brojevi zadani su rekurzivnom relacijom F(n) = F(n-1)+F(n-2), n >= 2; F(0) = 0; F(1) = 1;
Implementacija u C kôd glasi:

```
unsigned long Fib( unsigned n ) {
  if( n < 2 ) return (unsigned long)n;
  return Fib(n-1)+Fib(n-2);
}
```

Apriorna složenost algoritma je eksponencijalna: 2^n.

Opaska: uočite kako smo tipove za argument i povratnu vrijednost definirali kao unsigned! Ovo je korisno, jer Fib za negativne brojeve nije definiran, i svaki je Fib pozitivan, pa se ova odluka čini savršeno razumnom.

ZADATAK 3.

Napisati sva stanja polja u koracima uzlaznog Shell sorta ako je h_k niz {4,3,1}. Početno polje glasi:
3,12,2,5,7,4,9,8,10,1,6,11

Original polje:
03 12 02 05 07 04 09 08 10 01 06 11
Sort sa $h_k = 4$
03 04 02 05 07 12 09 08 10 01 06 11
03 04 02 05 07 01 09 08 10 12 06 11
03 01 02 05 07 04 09 08 10 12 06 11
03 01 02 05 07 04 06 08 10 12 09 11
Sort sa $h_k = 3$
03 01 02 05 07 04 06 08 10 12 09 11
Sort sa $h_k = 1$
01 03 02 05 07 04 06 08 10 12 09 11
01 02 03 05 07 04 06 08 10 12 09 11
01 02 03 05 04 07 06 08 10 12 09 11
01 02 03 04 05 07 06 08 10 12 09 11
01 02 03 04 05 06 07 08 10 12 09 11
01 02 03 04 05 06 07 08 10 09 12 11
01 02 03 04 05 06 07 08 09 10 12 11
01 02 03 04 05 06 07 08 09 10 11 12

ZADATAK 4.

U binarnom stablu su upisani brojevi prijava (četveroznamenkasti cijeli broj), imena studenata (do 20 znakova) i plasman na ispitu (četveroznamenkasti cijeli broj). Stablo je sortirano po broju prijave, a treba ga prepisati u drugo binarno stablo, sorturano po plasmanu. Lijevo je manji broj, a desno veći.

Slijedeći tekst zadatka, prvo moramo napisati kako izgleda struktura pojedinog čvora stabla. Nakon toga treba proči kroz kompletno stablo, i pri tome svaki stvoriti kopiju svakog čvora, te kopiju dodati u novo stablo. Za riješenje će nam u te svrhe trebati nekoliko pomočnih funkcija (možda može i drugačije, ali trenutno je ovo jedino što mi pada na pamet). Bacite pogled na riješenje, a zatim pogledajte ispod objašnjenje.

```
typedef struct CvorStr {
  CvorStr *Left;  // lijevo dijete
  CvorStr *Right; // desno dijete
  int BrojPrijave;
  char Ime[21];
  int Plasman;
} Cvor;

void Dodaj( Cvor *cv, Cvor **Glava ) {
  if( *Glava != NULL ) {
   if( cv->Plasman < (*Glava)->Plasman ) Dodaj( cv, &(*Glava)->Left );
   else Dodaj( cv, &(*Glava)->Right );
   return;
  }
  *Glava = cv;
  cv->Left = cv->Right = NULL;
}

int PrepisiRekurz( Cvor *Izvor, Cvor **Novo ) {
  Cvor *pom;
  if( Izvor == NULL ) return 0;        // ako nema trenutnog cvora, izlaz!
  pom = (Cvor*)malloc(sizeof(Cvor));  // inace prvo alocirajmo novi cvor, pa
  if( pom == NULL ) return 1;
  memcpy(pom, Izvor, sizeof(Cvor));  // prekopirajmo stari cvor u novi
  Dodaj( Novo, pom );
  if( PrepisiRekurz( Izvor->Left, Novo ) ) return 1;
  if( PrepisiRekurz( Izvor->Right, Novo ) ) return 1;
  return 0;
}

void IzbrisiStablo( Cvor *cv ) {
  if( cv == NULL ) return;
  Izbrisi( cv->Left );
  Izbrisi( cv->Right );
  free( cv );
}
Cvor *Prepis( Cvor *Glava ) {
  Cvor *cv;

  cv = NULL;
  if( PrepisiRekurz( Glava, &cv ) ) {
   IzbrisiStablo(cv);
   return NULL;
  }
  return cv;
}
```

Sam čvor binarnog stabla mislim da je definiran dovoljno jasno. No koja je ideja rješenja? Potrebno je napraviti novu kopiju binarnog stabla. To znači da moramo proči kroz svaki čvor zadanog binarnog stabla (enumeracija čvorova, sjećate se: inorder, preorder, postorder,...), i kopiju svakog čvora dodati u novo binarno stablo sortirano po novom uvjetu. Činjenica da je staro binarno stablo već uređeno na neki način ovdje ne igra nikakvu ulogu. Jedino što je važno jest proči kroz svaki čvor i njegovu kopiju dodati u novo stablo. No ovo su dva posla! I oba su rekurzivna! Prvo moramo napisati funkciju koja će proči kroz svaki čvor i pozvati funkciju Dodaj. Ovo može biti jedna od klasičnih enumeracija, i to je realizirano sa funkcijom **PrepisiRekurz**. Ona prvo provjeri ima li valjani čvor; ako nema, prekida sa izvođenjem (i to uspješno!). Inaće stvara kopiju čvora te poziva funkciju **Dodaj** koja novi čvor smješta u novo stablo. Ako alokacija ne uspije, vraća se broj 1 kao upozorenje za prekid svih daljnjih dodavanja. U slučaju da je sve u redu, funkcija se poziva rekurzivno sa svakim od svoje djece. Isto tako, ako dodavanje bilo kojeg djeteta ne uspije, funkcija će primiti 1, i proslijediti to svom pozivatelju kao opomenu. Ukoliko je pak sve dodano u redu, funkcija vraća 0. Eto, ovo je bio klasični primjer preorder enumeracije.

Pogledajmo sada kako je realizirana funkcija **Dodaj**. Ona je isto rekurzivna, iako se dodavanje može izvesti i u običnoj for petlji, dakle nerekurzivno. Prvo što funkcija provjerava jest jesmo li stigli na neki rub stabla (pa je roditeljev left ili right, ovisno kamo smo stigli, jednak NULL). Ako je ovo ispunjeno, tada se tu možemo ubaciti, i to činimo. Ukoliko ovo nije istina, znači da roditelj pokazuje na neki čvor sa kojim se moramo usporediti. Ako je naš plasman manji, rekurzivno pozivamo funkciju za dodavanje u lijevo, inaće se pokušavamo dodati desno. Uočite da dodavanje u binarno stablo uvijek uspijeva! Naime, ova funkcija ne alocira memoriju, i nema razloga zašto funkcija ne bi bila izvedena uspješno. Stoga funkcija niti ne vraća nikakvu vrijednost!

I konačno do glavne funkcije: **Prepis**! Funkcija prima samo jedan parametar: pokazivač na stablo koje treba prepisati. Funkcija definira pokazivač na novo stablo: cv, i postavlja ga na NILL jer je novo stablo još prazno. Zatim poziva funkciju PrepisiRekurz. Ukoliko funkcija PrepisiRekurz uspije prepisati sve članove, tada će vratiti 0 i uvjet if-a neće biti zadovoljen. To će za posljedicu imati izvršenje naredbe: return cv, čine se pozivatelju vraća ispravni pokazivač na novo stablo i funkcija se prekida. Ako je pak negdje u procesu stvaranja kopije došlo do greške, očito je da potpunu kopiju stabla nismo uspjeli kreirati. Zato pozivamo funkciju **IzbrisiStablo**, koja će pobrisati onaj dio koji smo uspjeli prekopirati, i vratiti pozivatelju NULL pokazivač kao indikaciju greške.

I opet smo došlo do neprecizno definiranog zadatka koji je rezultirao riješenjem od jedne strukture i ČETIRI funkcije! Riješenje zadatka moglo se je napisati puno jednostavnije da su bili poznati odgovori na pitanja:

- treba li pozivatelj imati indikaciju greške?
- ako se dogodi greška, kako postupiti? (obrisati do tada stvoreno, ili vrati što smo uspjeli stvoriti?)

Pitanja su i opet potpuno razumna, ali bez pravih odgovora, kao konačna riješenja mogu nastati cijeli programi za čiju je implementaciju potrebno ono što studenti imaju u najmanjoj mijeri - PUNO vremena!

Eto, ovo se je riješenje zbilja proteglo. No to se je dogodilo jer smo opet morali pretpostavljati. A ja kakav jesam, uvijek idem na sve moguće slučajeve i programe pišem krajnje obrambeno, što znači i puno, puno koda! No ako se nađete u ovakvoj situaciji, možete si olakšati život. Budući da nitko nije rekao što treba činiti ako dođe do greške, mogli ste bez problema zaboraviti na rukovanje njima, izbaciti funkciju ObrisiStablo i vratiti ono što dobijete - bez obzira je li to doista permutirana kopija izvornog stabla ili nije.

ZADATAK 5.

Jedan zapis tablice raspršenog adresiranja sadrži šifru (cijeli broj), ime duljine 25+1 znak i cijenu (realni broj) nekog artikla. Veličina zapisa na disku je BLOK. Prazni zapis sadrži cijenu jednaku nuli. Očekuje se do 1000 artikala, a tablica je dimenzionirana za oko 30% većom od punog kapaciteta. Napisati funkciju koja će pronaći koliko ima potpuno ispunjenih pretinaca.

Eh, opet vražije pretpostavke! A GDJE SE NALAZI DOTIČNA TABLICA? Mislim, očito u datoteci, ali to naša funkcija ne zna! JE LI DATOTEKA VEČ OTVORENA, pa funkcija prima FILE* na datoteku, ILI NIJE? Ako nije, KAKO SE ZOVE, da je mi sami otvorimo! Pretpostavka 1: datoteka nije otvorena, ime joj je "artikli.dat", i nalazi se u istom direktoriju kao i program koji će koristiti ovu funkciju.

Idemo sada definirati kako izgleda jedan zapis. Prvo imamo šifru - cijeli broj! Baš su nam puno rekli! Je li to možda long, ili je običan int? Opće je poznata stvar da šifra nema blage veze sa rednim brojem zapisa, tako da ništa ne znamo; naravno, osim činjenice da šifre mogu imati jako puno znamenaka - sjetite se samo onih bar-kodova na više manje svim artiklima u trgovinama. Pretpostavka broj 2: šifra je tipa long. Ajmo dalje. Ne bi čovjek vjerovao, ali ime je jasno definirano: 25 znakova plus jedan za nul-terminator. No pređimo ni na cijenu - realan broj! Opet velika pomoč, nema šta. Pretpostavka broj 3: cijena je tipa double. Ne zato što se u tom dučanu prodaju NASAine rakete pa nam treba malo širi opseg za cijene, već čisto iz praktičnih razloga: tvrdim da float tip ne može prikazati sve decimalne brojeve sa dvije decimale kao konačne brojeve, dok se sa double tipom bar približavamo ovoj apstrakciji. Eto tako, oboružani hrpom pretpostavki pokušajmo riješiti ovaj zadatak.

Kako pišemo funkciju, a ne program, ne smijemo izvan te funkcije definirati nekakve globalne varijable, tipove i sl., pa ćemo sav kod morati utrpati u tijelo jadne funkcije. Evo kako:

```
int PotpunoPuni( void ) {

  // definirajmo jednu od mogućih struktura zapisa
  typedef struct {
    long sifra;
    char ime[26];
    double cijena;
  } Element;

  // definirajmo si potrebne konstante uz napomenu da imena ovih konstanti odstupaju od
  // onih sa predavanja, ali su u savrsenom skladu sa imenima koja sam koristio u tekstu
  // o rasprsenom adresiranju na ovim stranicama.
  #define BPS = BLOK  // velicina sektora;
                      // ovo je zadano kao BLOK, a koristene su oznake kao u
                      // tekstu o Rasprsenom adresiranju na ovim stranicama
  #define BSL = 2     // odabiremo npr. 2 sektora po ladici
  #define BP = 1000   // zadano nam je da trebamo 1000 artikala pohraniti u tablicu
  #define BPL = BPS * BSL / sizeof(Element)  // ovoliko artikala stane u jednu ladicu
  #define FS = 1.3    // dodajemo 30% više mjesta od potrebnoga da smanjimo
vjerojatnost preljeva (zadano)
  #define UBL = (int)((double)BP / BPL * FS) // ukupan broj ladica u tablici

  // definirajmo jednu ladicu u koju cemo ucitavati podatke
  char cLadica[BPS*BSL];               // polje koje je veliko tocno kao jedna ladica
  FILE *f;    // dakako i pokazivac za datoteku
  int br;     // brojac potpuno praznih ladica
  int puna;   // i pomocni indikator koji kaze je li ladica potpuno prazna

  br = 0;     // postavimo brojac na nulu
  f = fopen("artikli.dat", "rb");  // otvorimo datoteku
  if( f == NULL ) return -1;       // greska? sorry!
  // inace, tako dugo dok uspijevamo procitati ladicu
  while( fread(cLadica,sizeof(cLadica),1,f) ) {
   puna = 1;    // pretpostavimo da je puna
   // ako nije puna, to ce se iskritalizirati u for petlji
   for( i = 0; i < BPL; i++ ) {
    if( cLadica[i].cijena == 0 ) { puna = 0; break; }
   }               // ako je puna, povecaj brojac za 1
   if( puna ) br++;            // i tako opet ispocetka
  }
  fclose(f);   return br;
}
```