Programiranje 2

Dinamičke strukture podataka — Liste

Milena Vujošević Janičić
Jelena Graovac

www.matf.bg.ac.rs/~milena
www.matf.bg.ac.rs/~jgraovac

Matematički fakultet

Pregled

Dinamičke strukture podataka

- Dinamička alokacija memorije omogućava građenje specifičnih, dinamičkih struktura podataka koje su u nekim situacijama pogodnije od statičkih zbog svoje fleksibilnosti.
- Najznačajnije dinamičke strukture podataka su povezane liste i stabla (čiji elementi mogu da sadrže podatke proizvoljnog tipa).

Nizovi

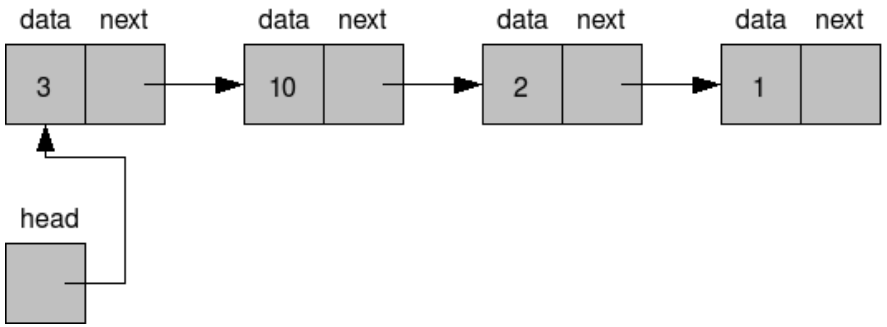
Pre nego što nastavimo sa listama i stablima, potsetimo se nekih osobina nizova.
Koliko je efikasan proces ...

- ... modifikacije jednog elementa niza?
- ... dodavanja elementa na kraj niza?
- ... brisanja elementa sa kraja niza?
- ... dodavanja elementa na početak niza?
- ... brisanja elementa sa početka niza?
- ... dodavanja elementa u sredini niza?
- ... brisanja elementa iz sredine niza?

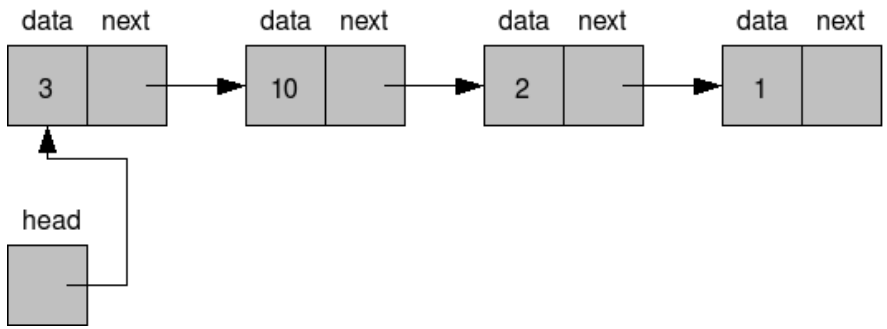
Pregled

Liste

- Povezanu listu čine elementi koji sadrže podatke izabranog tipa i pokazivače.
- Svaki pokazivač pokazuje na jedan (sledeći) element liste i ti pokazivači povezuju elemente u jednu celinu — u listu.



Liste



- Prvi cvor u sekvenci naziva se glava liste.
- Ostatak liste (bez glave) je takodje lista, i naziva se rep liste.
- Lista koja ne sadrzi cvorove naziva se prazna lista.
- Poslednjem elementu u listi pokazivač na sledeći element ima vrednost NULL.

Zašto praviti liste?

- Liste se upotrebljavaju za čuvanje podataka (slično kao i nizovi).
- Prednosti liste u odnosu na statički/dinamički niz: efikasno se dodaju i brišu elementi (konstantna složenost).
- Mane liste u odnosu na statički/dinamički niz: pristup n-tom elementu liste je linearne složenosti.
- Najbolji izbor strukture podataka (niz statički/dinamički ili lista) je vezan za specifičnosti konkretnog problema i najvažnije zahteve.

Izbor

Koju strukturu podataka koristiti ukoliko je potrebno da...

- ... često pristupamo podacima i menjamo njihovu vrednost?
- ... često dodajemo nove podatke na kraj strukture?
- ... često dodajemo nove podatke na početak strukture?
- ... često dodajemo nove podatke na odgovarajuće mesto (tj bilo gde)?
- ... često dodajemo i brišemo podatke?

Čvor liste

```
typedef struct cvor {
  <type> podatak;
  struct cvor *sledeci;
} Cvor;
```

- <type> označava tip podatka koji element liste sadrži. To može biti proizvoljan tip (moguće i struktura).
- Element može da sadrži i više od jednog podatka:

```
typedef struct cvor {
  <type1> podatak1;
  ...
  <typeN> podatakN;
  struct cvor *sledeci;
} Cvor;
```

Kako kreirati listu?

- Listi se pristupa preko pokazivača na početak i/ili kraj liste.
- Lista je najpre prazna, zatim se u praznu listu dodaju elementi.
- Dodavanje elemenata u listu može se vršiti dodavanjem elemenata na početak, na kraj ili negde u sredinu liste.
- Pre nego što se element dodaje u listu, potrebno ga je kreirati.

Kreiranje elemenata liste

Kreiranje elementa liste vrši se dinamičkom alokacijom. Nakon dinamičke alokacije potrebnog prostora, neophodno je izvršiti i inicijalizaciju vrednosti kreiranog čvora:

```
typedef struct cvor {
    int vrednost; /* podatak koji cvor sadrzi */
    struct cvor *sledeci; /* pokazivac na sledeci cvor */
} Cvor;

Cvor *napravi_cvor(int broj) {
    Cvor *novi = (Cvor *) malloc(sizeof(Cvor));
    if(novi == NULL) return NULL; /*Signal da alokacija nije uspela*/

    novi->vrednost = broj;
    novi->sledeci = NULL;

    return novi;
}
```

Dodavanje elementa u listu — na početak

- Najjednostavnije je dodati element na početak liste.
- Najpre je potrebno kreirati čvor i u njega smestiti odgovarajuće podatke.
- Zatim je potrebno povezati kreirani čvor sa početkom liste i promeniti da glava liste sada bude pokazivač na novorekirani čvor.
- Dodavanjem elementa na početak liste menja se glava liste.

Dodavanje elementa u listu — na početak

- Listu možemo formirati uzastopnim dodavanjem elemenata na njen početak (dakle višestrukim pozivima funkcije `dodaj_na_pocetak_liste`).

```
int dodaj_na_pocetak_liste(Cvor **adresa_glave, int broj) {
    Cvor *novi = napravi_cvor(broj);
    if(novi == NULL) return 1;

    /* uvezujemo novi cvor na pocetak */
    novi->sledeci = *adresa_glave;

    /*Postavljamo novu glavu liste*/
    *adresa_glave = novi;

    return 0;
}
```

Dodavanje elementa u listu — na početak

- Pre nego što nastavimo dalje, pogledajmo jedan primer kreiranja liste celih pozitivnih brojeva. Korisnik brojeve unosi sa standardnog ulaza sve dok ne unese -1.

```
int main() {
  Cvor *glava = NULL;
  int broj;
  scanf("%d", &broj);
  while(broj>=0) {
    if(dodaj_na_pocetak_liste(&glava, broj) == 1) {
      fprintf(stderr, "Greska: Neuspesna alokacija memorije za cvor.\n");
      oslobodi_listu(&glava);
      exit(EXIT_FAILURE);
    }
  scanf("%d", &broj);
  }
  /* obrada liste */
  oslobodi_listu(&glava);
  exit(EXIT_SUCCESS);
}
```

Prolazak kroz listu

- Transformacija svih elemenata liste obuhvata naredne korake:
  1. Postavljanje jednog pokazivača (nazovimo ga „tekuci“) tako da pokazuje na početak liste;
  2. Proveru da li pokazivač „tekuci“ pokazuje na praznu listu, ukoliko je lista prazna, završava se proces transformacije;
  3. Transformaciju podatka u čvoru na koji pokazuje pokazivač „tekuci“;
  4. Pomeranje pokazivača „tekuci“ tako da pokazuje na naredni čvor liste, i povratak na korak 2.

Prolazak kroz listu

Prolazak kroz listu — transformacija svih elementa.

```
void uvecaj_za_tri(Cvor *glava) {
  Cvor *tekuci;
  for (tekuci = glava; tekuci != NULL; tekuci = tekuci->sledeci)
    tekuci->vrednost += 3;
}
```

Prolazak kroz listu

- Rekurzivna definicija liste daje osnovu za rekurzivne implementacije algoritma za rad sa listama.
- Prateći rekurzivnu definiciju liste, transformacija elemenata liste se može definisati na sledeći način:
  - Ukoliko je lista prazna, posao je završen (jer nemamo šta da transformišemo);
  - Ukoliko lista nije prazna, transformiši glavu liste a potom pozovi rekurzivno istu funkciju koja kao argument prima rep liste.

Prolazak kroz listu

Rekurzivni prolazak kroz listu — transformacija svih elementa.

```
void uvecaj_za_tri(Cvor *glava) {
  if(glava == NULL)
    return;
  glava->vrednost+=3;
  uvecaj_za_tri(glava->sledeci);
}
```

Obrada liste

- Rekurzivna obrada liste uvek se sastoji iz dva koraka:
  1. Slučaj kada je lista prazna;
  2. Slučaj kada lista nije prazna i kada se obrađuje glava liste, a zatim rekurzivno poziva funkcija čiji je argument rep liste.

Prolazak kroz listu

Prolazak kroz listu — pronalaženje određenog elementa.

```
Cvor *pretrazi_listu(Cvor *glava, int broj) {
  Cvor *tekuci;
  for (tekuci = glava; tekuci != NULL; tekuci = tekuci->sledeci)
    if (tekuci->vrednost == broj)
      return tekuci;

  return NULL;
}
```

Za vežbu uradite rekurzivnu varijantu ove funkcije.

Dodavanje elementa u listu — na kraj

- Dodavanje elemenata na kraj liste — najpre je potrebno odrediti kraj liste.
- Kraj liste se može odrediti prolaskom kroz listu i pronalaženjem elementa čiji je sledeći element NULL.
- Ukoliko postoji potreba za čestim dodavanjem na kraj liste, može se čuvati pokazivač na kraj liste.

Prolazak kroz listu — pronalaženje poslednjeg elementa

```
Cvor *pronadji_poslednji(Cvor *glava) {
  if(glava == NULL) return NULL;

  while (glava->sledeci != NULL)
    glava = glava->sledeci;

  return glava;
}
```

Za vežbu napisati rekurzivnu varijantu ove funkcije.

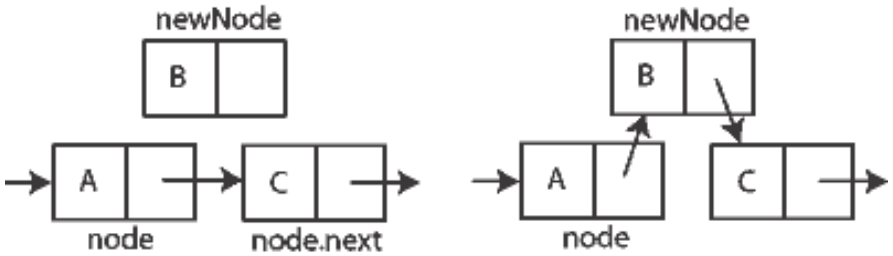
Dodavanje elementa u listu — na kraj

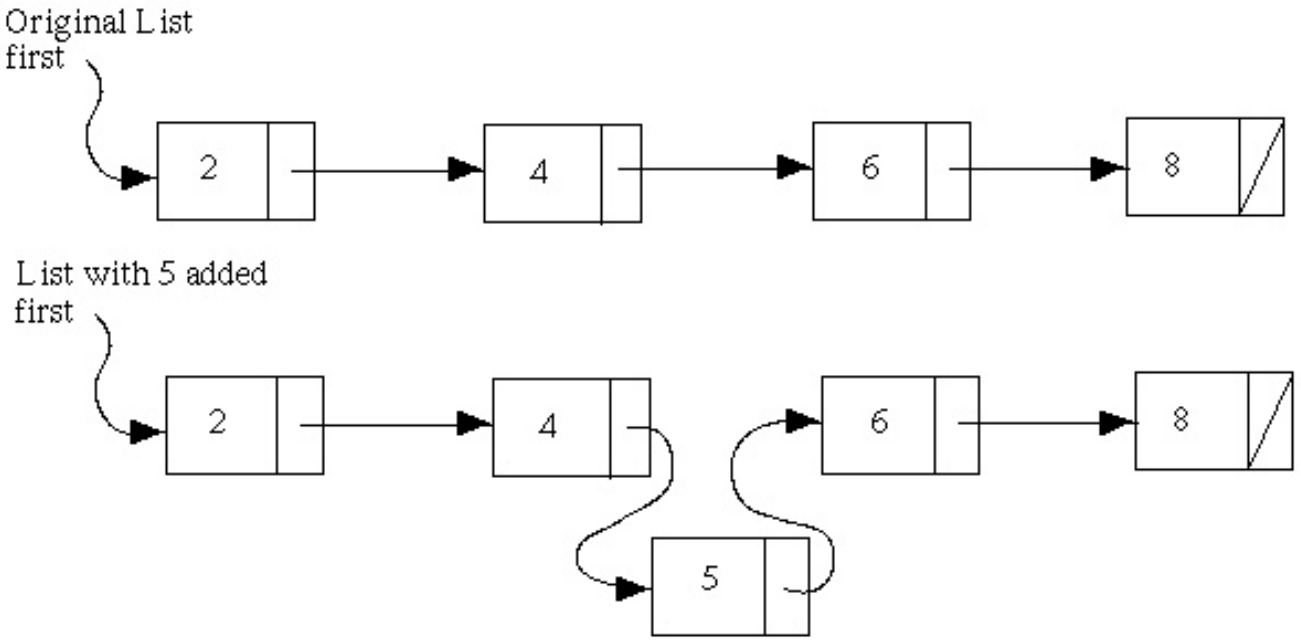
- Pretpostavimo da smo pronašli pokazivač na poslednji element u listi. Tada se dodavanje na kraj svodi na:
  - slučaj ukoliko je lista prazna — glava liste postaje element koji dodajemo
  - slučaj ukoliko lista nije prazna — pokazivač poslednjeg elementa se menja tako da ukazuje na element koji dodajemo
- Dodavanjem na kraj može se promeniti glava liste.

Dodavanje elementa u listu — na kraj

```
int dodaj_na_kraj_liste(Cvor **adresa_glave, int broj) {
  Cvor *novi = napravi_cvor(broj);
  if(novi == NULL) return 1;

  if(*adresa_glave == NULL) {
    *adresa_glave = novi;
  } else {
    Cvor *poslednji = pronadji_poslednji(*adresa_glave);
    poslednji->sledeci = novi;
  }

  return 0;
}
```

Dodavanje elementa u listu — u sredinu





Dodavanje elementa u sortiranu listu

- Napravi čvor koji želiš da umetneš
- Ako je lista prazna, onda je nova lista čvor koji smo napravili
- Ako lista nije prazna, proveri da li čvor treba da se umetne na početak
- Ukoliko ne umećemo na početak, pronađi mesto tj čvor A iza kojeg je potrebno umetnuti novokreirani čvor B
- Prvo poveži novi čvor B sa narednim čvorom od čvora A u listi
- Zatim poveži čvor A sa čvorom B
- Umetanjem u sredinu može se izmeniti glava liste

Dodavanje elementa u sortiranu listu

```
int dodaj_sortirano(Cvor **adresa_glave, int broj) {
  if(*adresa_glave == NULL){
    Cvor *novi = napravi_cvor(broj);
    if(novi == NULL) return 1;

    *adresa_glave = novi;
    return 0;
  }

  if((*adresa_glave)->vrednost >= broj) {
    return dodaj_na_pocetak_liste(adresa_glave, broj);
  }

  Cvor *pomocni = pronadji_mesto_umetanja(*adresa_glave, broj);

  return dodaj_iza(pomocni, broj);
  }
```

Dodavanje elementa u sortiranu listu

```
Cvor *pronadji_mesto_umetanja(Cvor *glava, int broj) {
  if(glava == NULL) return NULL;

  while(glava->sledeci != NULL && glava->sledeci->vrednost < broj)
    glava = glava->sledeci;

  return glava;
}

int dodaj_iza(Cvor *tekuci, int broj) {
  Cvor *novi = napravi_cvor(broj);
  if(novi == NULL) return 1;

  novi->sledeci = tekuci->sledeci;
  tekuci->sledeci = novi;

  return 0;
}
```

Brisanje prvog elemenata liste

- Brisanje prvog elementa liste:
  - Ukoliko lista nije prazna, sačuvaj pokazivač na sledeći element
  - Obriši glavu
  - Nova glava liste je sačuvani pokazivač
- Brisanjem prvog elemena liste, menja se glava liste.

Brisanje prvog elementa iz liste

```
void obrisi_prvi(Cvor **adresa_glave) {
  Cvor *pomocni = NULL;
  if(*adresa_glave == NULL)
    return;
  pomocni = (*adresa_glave)->sledeci;
  free(*adresa_glave);
  *adresa_glave = pomocni;
}
```

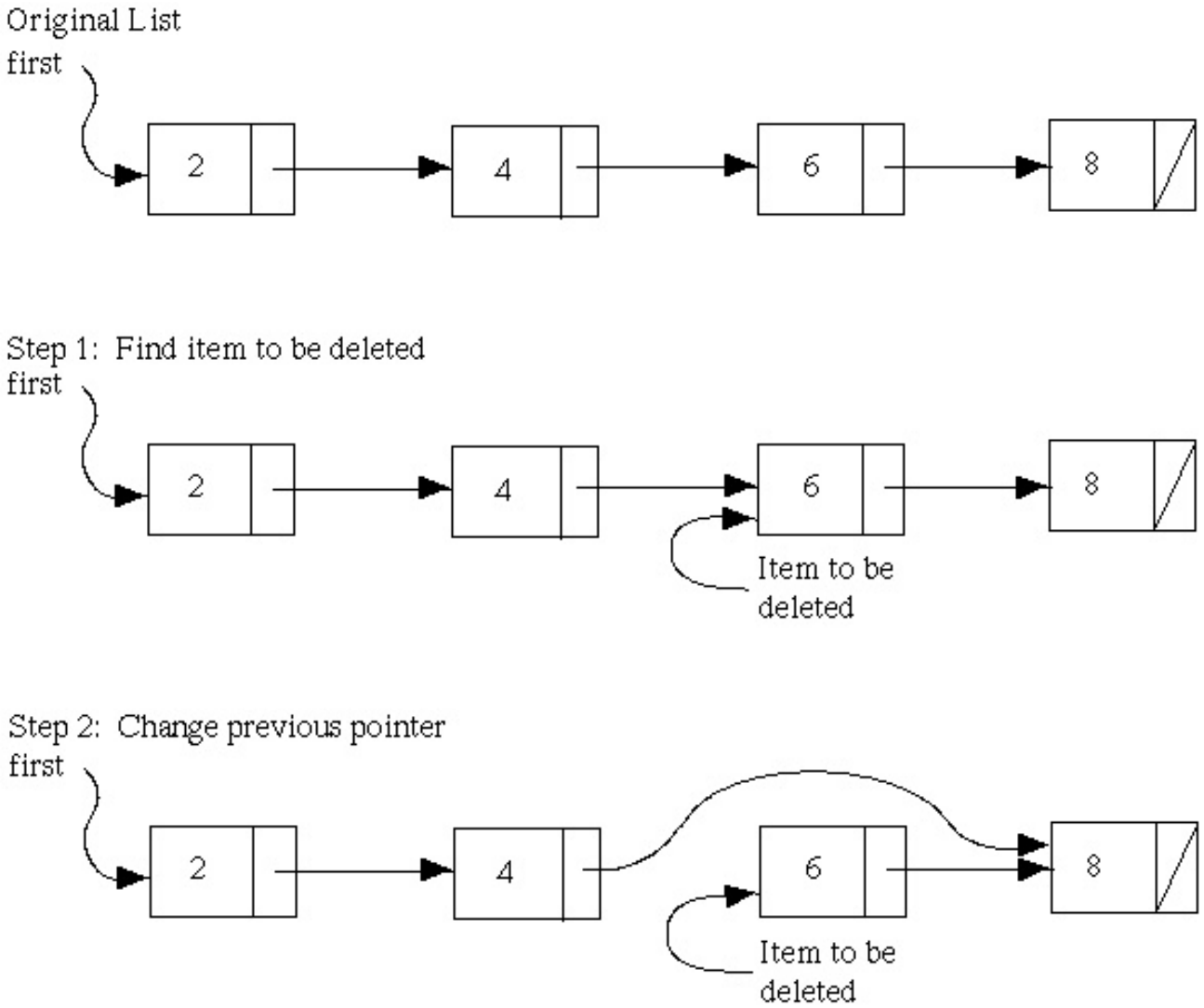
Brisanje cele liste

- Oslobađanje memorije koju zauzima cela lista je uzastopno brisanje prvog elementa u listi

Brisanje cele liste

```
void oslobodi_listu(Cvor **adresa_glave) {
  Cvor *pomocni = NULL;

  while (*adresa_glave != NULL) {
    pomocni = (*adresa_glave)->sledeci;
    free(*adresa_glave);
    *adresa_glave = pomocni;
  }
}
```

Brisanje jednog elementa iz liste



Brisanje elemenata sa zadatom vrednošću iz liste

- Brisanje elemenata sa zadatom vrednošću iz liste se sastoji od narednih koraka
  - Ukoliko je lista prazna, posao je završen
  - Proveri da li se element koji treba da se obriše nalazi (jednom ili više puta) na početku liste, ako je tako primeni postupak brisanja elementa sa početka liste
  - Pronađi pokazivač na element u listi iza kojeg se nalazi pokazivač na element koji želiš da obrišeš.
  - Sačuvaj vrednost pokazivača na element koji želiš da obrišeš
  - Prespoji elemente u listi tako da element koji želimo da obrišemo više nije u listi
  - Oslobodi memoriju koju zauzima element koji želimo da obrišemo.

Brisanje elemenata sa zadatom vrednošću iz liste

```
void obrisi_cvor(Cvor **adresa_glave, int broj) {
  Cvor *tekuci = NULL;
  Cvor *pomocni = NULL;

  /* Sa pocetka liste se brisu svi cvorovi koji su jednaki
     datom broju i azurira se pokazivac na glavu liste */
  while (*adresa_glave != NULL && (*adresa_glave)->vrednost == broj){
    pomocni = (*adresa_glave)->sledeci;
    free(*adresa_glave);
    *adresa_glave = pomocni;
  }

  /* Ako je nakon ovog brisanja lista prazna, to je kraj */
  if (*adresa_glave == NULL) return;
```

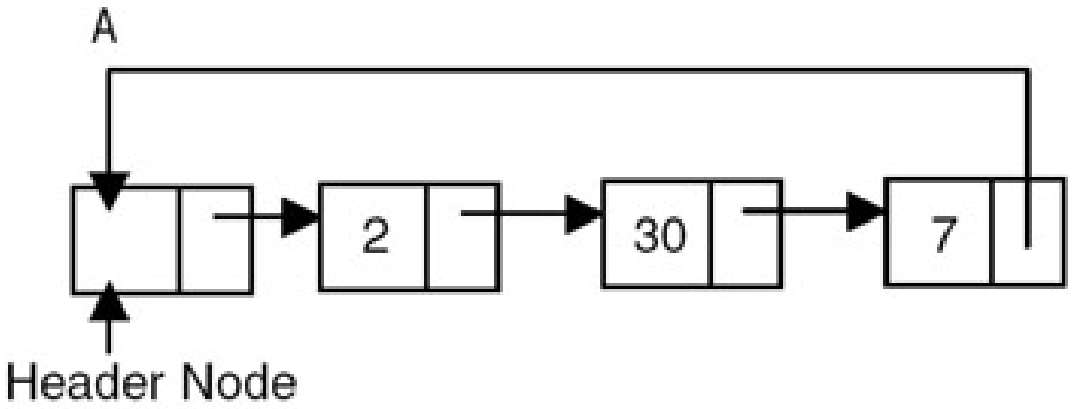
Brisanje elemenata sa zadatom vrednošću iz liste

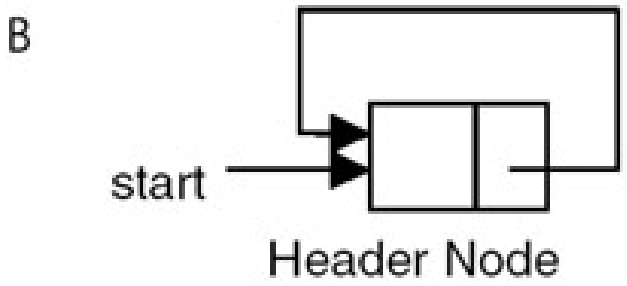
```
/* Od ovog trenutka, u svakoj iteraciji petlje promenljiva tekuci
   pokazuje na cvor cija je vrednost razlicita od trazenog broja.
   Isto vazi i za sve cvorove levo od tekuceg. Poredi se vrednost
   sledeceg cvora (ako postoji) sa trazenim brojem. Cvor se brise
   ako je jednak, a ako je razlicit, prelazi se na sledeci. Ovaj
   postupak se ponavlja dok se ne dodje do poslednjeg cvora. */
  tekuci = *adresa_glave;
  while (tekuci->sledeci != NULL)
    if (tekuci->sledeci->vrednost == broj) {
      pomocni = tekuci->sledeci;
      tekuci->sledeci = pomocni->sledeci;
      free(pomocni);
    } else {
      tekuci = tekuci->sledeci;
    }
  return;
}
```

Kružne liste

- Kružne liste su liste u kojoj poslednji element pokazuje na početak liste
- To omogućava da se od bilo kog elementa u listi stigne do svakog drugog elementa u listi, čime je izbor prvog i poslednjeg elementa u listi relativan
- Kružne liste se mogu koristiti za rešavanje raznih vrsta problema

Kružne liste





Kružne liste

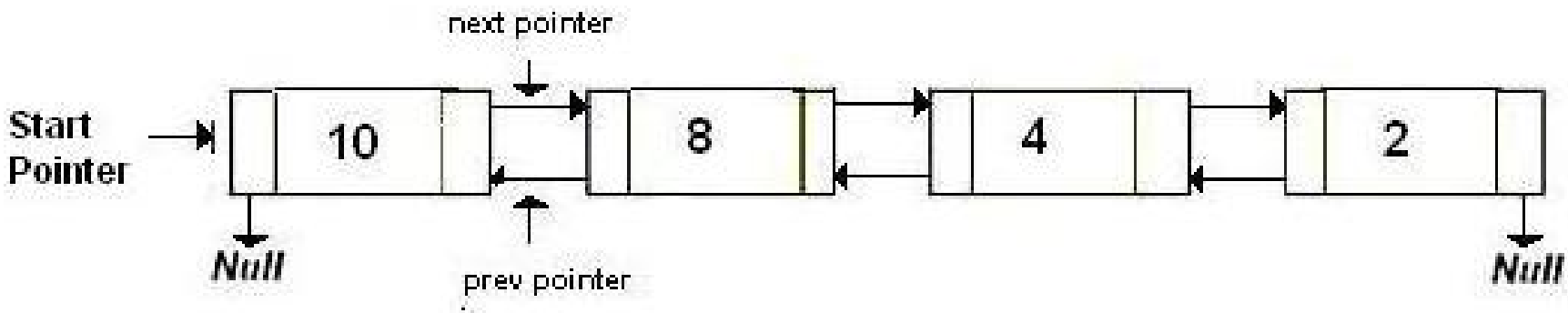
Prolazak kroz kružnu listu:

```
void uvecaj_za_tri(Cvor *glava) {
  Cvor *tekuci = glava;
  if (tekuci == NULL) return;
  do {
    tekuci->vrednost += 3;
    tekuci = tekuci->sledeci;
  } while(tekuci != glava);
}
```

Dvostruko povezane liste

- Svaki element u (jednostruko) povezanoj listi ima jedan pokazivač — pokazivač na sledeći element liste.
- Zato je listu jednostavno obilaziti u jednom smeru ali je vremenski zahtevno u suprotnom.
- U dvostruko povezanoj listi, svaki element sadrži dva pokazivača — jedan na svog prethodnika, a drugi na svog sledbenika.
- Dvostruko povezana lista omogućava jednostavno kretanje unapred i unazad kroz listu.
- Dodavanje i brisanje elemenata iz dvostruko povezane liste obuhvata održavanje vrednosti oba pokazivača.
- Mogu se koristiti i dvostruko povezane kružne liste.

Dvostruko povezane liste



Dvostruko povezane liste

```
typedef struct cvor{
  int vrednost;
  struct cvor *sledeci;
  struct cvor *prethodni;
} Cvor;
```

Dvostruko povezane liste

```
Cvor* napravi_cvor(int broj) {
  Cvor *novi  = (Cvor*)malloc(sizeof(Cvor));
  if(novi == NULL) return NULL;

  /* Inicijalizacija polja strukture */
  novi->vrednost = broj;
  novi->sledeci = NULL;
  /* Inicijalizujemo i drugi pokazivac! */
  novi->prethodni = NULL;

  return novi;
}
```

Dvostruko povezane liste

- Funkcije za rad sa dvostruko povezanim listama moraju da vode računa o oba pokazivača.
- I ovde je veoma bitan redosled postavljanja ovih pokazivača

Dvostruko povezane liste — dodavanje na početak

```
int dodaj_na_pocetak_liste(Cvor **adresa_glave, Cvor **adresa_kraja,
                           int broj) {
  Cvor *novi = napravi_cvor(broj);
  if (novi == NULL) return 1;

  /* Sledbenik novog cvora je glava stare liste */
  novi->sledeci = *adresa_glave;

  /* Ako stara lista nije bila prazna, onda prethodni cvor glave
  treba da bude novi cvor. Inace, novi cvor je u isto vreme i
  pocetni i krajnji. */
  if (*adresa_glave != NULL)
    (*adresa_glave)->prethodni = novi;
  else
    *adresa_kraja = novi;

  /* Novi cvor je nova glava liste */
  *adresa_glave = novi;

  return 0;
}
```

Dvostruko povezane liste — dodavanje na kraj

```
int dodaj_na_kraj_liste(Cvor **adresa_glave, Cvor **adresa_kraja,
                        int broj) {
  Cvor *novi = napravi_cvor(broj);
  if (novi == NULL) return 1;

  /* U slucaju prazne liste, glava nove liste je upravo novi cvor i
     ujedno i cela lista. Azuriraju se vrednosti na koje pokazuju
     adresa_glave i adresa_kraja. Ako lista nije prazna, novi cvor
     se dodaje na kraj liste kao sledbenik poslednjeg cvora i azurira
     se samo pokazivac na kraj liste */
  if (*adresa_glave == NULL) {
    *adresa_glave = novi;
    *adresa_kraja = novi;
  } else {
    (*adresa_kraja)->sledeci = novi;
    novi->prethodni = (*adresa_kraja);
    *adresa_kraja = novi;
  }

  return 0;
}
```

Dvostruko povezane liste — dodavanje u sortiranu listu

```
int dodaj_sortirano(Cvor **adresa_glave, Cvor **adresa_kraja, int broj) {
  if (*adresa_glave == NULL) {
    Cvor *novi = napravi_cvor(broj);
    if (novi == NULL) return 1;
    *adresa_glave = novi;
    *adresa_kraja = novi;
    return 0;
  }

  if ((*adresa_glave)->vrednost >= broj) {
    return dodaj_na_pocetak_liste(adresa_glave, adresa_kraja, broj);
  }

  Cvor *pomocni = pronadji_mesto_umetanja(*adresa_glave, broj);
  if (dodaj_iza(pomocni, broj) == 1) return 1;

  if(pomocni == *adresa_kraja)
    *adresa_kraja = pomocni->sledeci;

  return 0;
}
```

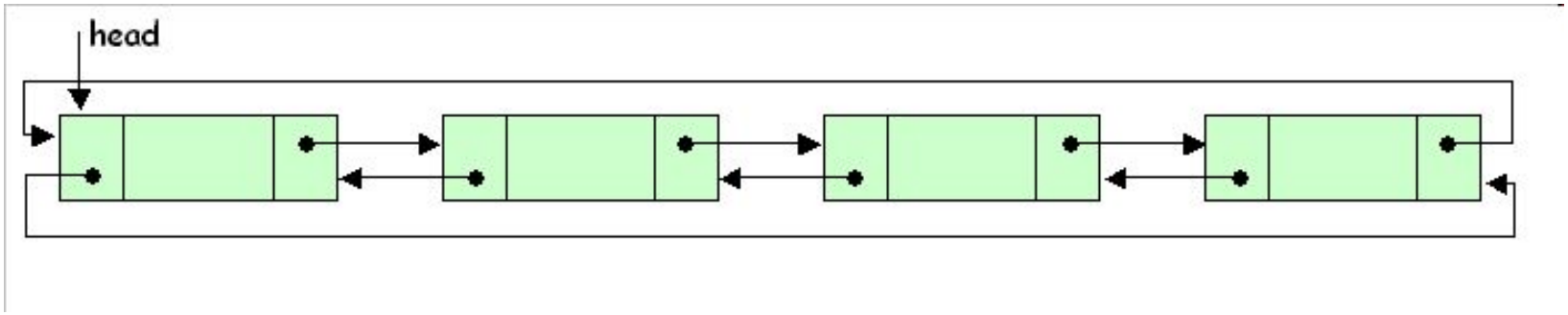
Dvostruko povezane liste — dodavanje u sortiranu listu

```
Cvor *pronadji_mesto_umetanja(Cvor *glava, int broj) {
  if(glava == NULL) return NULL;

  while(glava->sledeci != NULL && glava->sledeci->vrednost < broj)
    glava = glava->sledeci;

  return glava;
}

int dodaj_iza(Cvor *tekuci, int broj) {
  Cvor *novi = napravi_cvor(broj);
  if (novi == NULL) return 1;
  novi->sledeci = tekuci->sledeci;
  novi->prethodni = tekuci;

  if (tekuci->sledeci != NULL)
    tekuci->sledeci->prethodni = novi;
  tekuci->sledeci = novi;

  return 0;
}
```

Dvostruko povezane liste — brisanje

```
void obrisi_tekuci(Cvor **adresa_glave, Cvor **adresa_kraja, Cvor *tekuci) {
  /* Ako je tekuci NULL pokazivac, nema potrebe za brisanjem */
  if(tekuci == NULL) return;

  if(tekuci->prethodni != NULL)
    tekuci->prethodni->sledeci = tekuci->sledeci;

  if(tekuci->sledeci != NULL)
    tekuci->sledeci->prethodni = tekuci->prethodni;

  if(tekuci == *adresa_glave)
    *adresa_glave = tekuci->sledeci;

  /* Ako je cvor koji se brise poslednji u listi, azurira se i
     pokazivac na kraj liste */
  if(tekuci == *adresa_kraja)
    *adresa_kraja = tekuci->prethodni;

  free(tekuci);
}
```

Dvostruko povezane kružne liste



Pregled

Stek

- Stek (engl. stack) je struktura koja funkcioniše na principu LIFO (*last in, first out*).
- Stek ima sledeće dve osnovne operacije (koje treba da budu složenosti $O(1)$):

  potisni na stek (engl. push) dodaje element na vrh steka;
  skini sa steka (engl. pop) skida element sa vrha steka.
- Primeri steka u našem okruženju
- Stek se može implementirati na različite načine

Stek

- Stek se može implementirati korišćenjem nizova
- Kao vrh steka, pamti se trenutno poslednji element u nizu
- potisni na stek — dodaj na kraj niza
- skini sa steka — uzmi poslednji element niza

Stek

- Implementacija steka preko listi
- Funkcija `potisni_na_stek` je funkcija `dodaj_na_pocetak_liste`.
- Funkcija `skini_sa_steka` je modifikovana funkcija `obrisi_sa_pocetka_liste`.
- Funkcija `skini_sa_steka`, osim brisanja, treba da vrati i vrednost koja se nalazi u čvoru koji je u vrhu liste, kao i informaciju da li je skidanje sa steka uspelo (sa praznog steka ne možemo ništa skinuti).

Stek

```
int potisni_na_stek(Cvor **adresa_vrha, int broj) {
  Cvor *novi = napravi_cvor(broj);
  if (novi == NULL) return 1;
  novi->sledeci = *adresa_vrha;
  *adresa_vrha = novi;
  return 0;
}

int skini_sa_steka(Cvor **adresa_vrha, int *adresa_broja) {
  Cvor *pomocni = NULL;
  if (*adresa_vrha == NULL) return 1;

  pomocni = *adresa_vrha;
  if (adresa_broja != NULL)
    *adresa_broja = pomocni->vrednost;
  *adresa_vrha = pomocni->sledeci;
  free(pomocni);

  /* Vraca se indikator uspesno izvrsene radnje */
  return 0;
}
```

Pregled

Red

- Red (engl. queue) je struktura koja funkcioniše na principu FIFO (*first in, first out*).
- Red ima sledeće dve osnovne operacije (koje treba da budu složenosti $O(1)$):

  dodaj u red (engl. add) dodaje element na kraj reda;
  skini sa reda (engl. get) skida element sa početka reda.
- Primeri reda u našem okruženju
- Red se može implementirati na različite načine

Red — niz

- I red se može implementirati korišćenjem niza
  - dodaj u red — dodaj na kraj niza
  - skini sa reda — uzmi sa početka niza
- Da bi uzimanje bilo efikasno, prave se *ciklični* nizovi
- Sledeći indeks se računa po modulu veličine rezervisane memorije n, tj naredni element (bilo da se dodaje na kraj ili skida sa početka) se računa sa $i = (i + 1) \mod n$
- Za indekse početka p i kraja k reda ne mora da važi da je $p < k$, ali je važno voditi računa da se sa krajem niza ne pregazi početak niza

Red — lista

- Ukoliko se za implementaciju reda koristi lista, da bi se ostvarilo efikasno dodavanje, pored pokazivača na početak reda čuva se i pokazivač na kraj reda.
- Zato funkcije koje rade sa redom često dobijaju oba ova pokazivača, na primer

```
int dodaj_u_red(Cvor **adresa_pocetka, Cvor **adresa_kraja, int broj);
int skini_sa_reda(Cvor **adresa_pocetka, Cvor **adresa_kraja,
                  int *adresa_broja);
```

Red — lista

```
int dodaj_u_red(Cvor **adresa_pocetka, Cvor **adresa_kraja,
                int broj);
```

- Dodavanjem elementa u red mogu se izmeniti i početak i kraj reda
- Ukoliko je red prazan, menjaju se oba, ukoliko nije, menja se samo kraj
- Povratna vrednost govori da li je dodavanje u red uspelo
- Prolazak kroz red ostaje isti kao i za običnu listu

Red — lista

```
int skini_sa_reda(Cvor **adresa_pocetka,
                  Cvor **adresa_kraja, int *adresa_broja);
```

- Brisanjem elementa iz reda mogu se izmeniti i početak i kraj reda
- Ukoliko red sadrži samo jedan element menjaju se oba, ukoliko red sadrži više elemenata, menja se samo početak
- Prilikom skidanja elementa sa reda, osim brisanja elementa iz liste, u treći argument funkcije se upisuje i skinuta vrednost
- Povratna vrednost govori da li je skidanje elementa sa reda uspelo (ako je red prazan, skidanje ne može da uspe)

Red

```
int dodaj_u_red(Cvor **adresa_pocetka, Cvor **adresa_kraja, int broj) {
  Cvor *novi = napravi_cvor(broj);
  if(novi == NULL) return 1;

  if(*adresa_kraja != NULL) {
    (*adresa_kraja)->sledeci = novi;
    *adresa_kraja = novi;
  } else {
    *adresa_pocetka = novi;
    *adresa_kraja = novi;
  }

return 0;
}
```

Red

```
int skini_sa_reda(Cvor **adresa_pocetka, Cvor **adresa_kraja,
                  int *adresa_broja) {
  Cvor *pomocni = NULL;
  if (*adresa_pocetka == NULL) return 1;

  if (adresa_broja != NULL)
    *adresa_broja = (*adresa_pocetka)->vrednost;

  pomocni = *adresa_pocetka;
  *adresa_pocetka = (*adresa_pocetka)->sledeci;
  free(pomocni);

  if (*adresa_pocetka == NULL)
    *adresa_kraja = NULL;

  return 0;
}
```

Red — kružna lista

- Red se može implementirati korišćenjem kružne liste
- U tom slučaju se za red pamti samo pokazivač na poslednji element u listi
- Iza poslednjeg elementa u kružnoj listi, nalazi se prvi element reda
- Dodavanje i uzimanje:
  - dodaj u red — dodavanje se vrši iza poslednjeg elementa, i ažurira se vrednost pokazivača na poslednji element
  - skini sa reda — uzimanje se vrši uzimanjem prvog elementa iza poslednjeg

Pregled

Literatura

- Slajdovi su pripremljeni na osnovu sedmog poglavlja knjige Predrag Janičić, Filip Marić: Programiranje 2
- Za pripremu ispita, slajdovi nisu dovoljni, neophodno je koristiti knjigu!