Strukture podataka

Autori:
Goran Savić
Milan Segedinac

1. Uvod

Za efikasno upravljanje podacima, neophodno je da podatke organizujemo. Podaci se mogu organizovati na različite načine. Ovo različiti načini organizacije se nazivaju **strukture podataka**. Najjednostavnija struktura podataka sa kojom smo se do sada sretali je promenljiva primitivnog tipa. Ova struktura skladišti jednu vrednost. Druga struktura podataka koju smo sretali je objekat koji omogućuje grupisanje podataka o određenom entitetu, kao i procedura koje definišu operacije nad tim podacima.

U nastavku ćemo se baviti komplikovanijim strukturama podataka koje omogućuju organizaciju više podataka, bilo da su oni primitivnog tipa ili da je reč o objektima. Primer strukture ovog tipa sa kojom smo se do sada sretali je bio niz. Niz omogućuje grupisanje podataka istog tipa.

Generalno, strukture podataka možemo klasifikovati po različitim kriterijumima. Često se strukture podataka klasifikuju kao uređene ili neuređene. Strukture podaaka koje su uređene, elemente evidentiraju u sekvenci. Sa druge strane, postoje neuređene strukture podataka kod kojih elementi nisu organizovani u sekvencu. To znači da kod ovakvih struktura ne postoji redosled elemenata (ne postoji pojam sledećeg ili prethodnog elementa u odnosu na određeni element).

Za analizu efikasnosti određene strukture podataka, najčešće su nam dva svojstva od interesa. Prvo svojstvo je brzina pronalaženja određenog elementa u strukturi (indeksiranje elementa). Drugo važno svojstvo je fleksibilnost strukture podataka. Ovo svojstvo određuje da li je i koliko jednostavno moguće vršiti dinamičku izmenu strukture (npr. povećanje veličine, izbacivanje nekog elementa itd.)

U zavisnosti od pomenutih svojstava, različite su performanse izvršavanja operacija nad elementima u strukturi. Standardne operacije su pronalaženje elementa, dodavanje novog elementa, brisanje elementa i izmena postojećeg elementa u strukturi.

U nastavku ćemo predstaviti neke često korišćene strukture podataka.

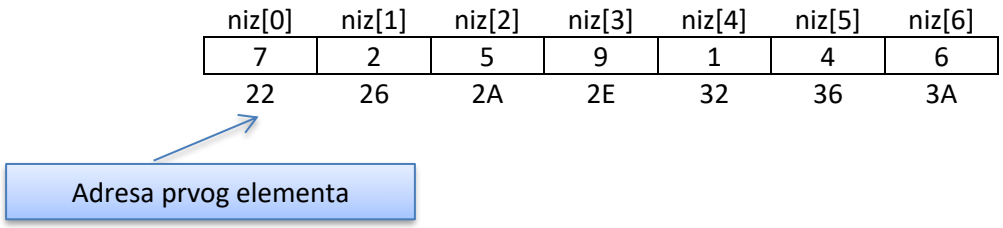
2. Sekvencijalne strukture podataka

Sekvencijalne strukture podataka organizuju elemente u sekvencu u kojoj između elemenata postoji redosled. Dakle, određeno je koji elementi prethode i slede određenom elementu. Prva struktura ovog tipa o kojoj će ovde biti reči je niz.

Niz

Niz je uređena kolekcija elemenata istog tipa. Većina programskih jezika niz realizuje tako da se svi elementi niza skladište u uzastopnim memorijskim lokacijama. Ovakav način skladištenja ima za posledicu da je indeksiranje elementa u nizu vrlo efikasno. Ako uskladištimo adresu prvog elementa u nizu, onda se do bilo kojeg drugog elementa u nizu dolazi dodavanjem određenog broja memorijskih lokacija na tu adresu. Na primer, ako imamo niz celobrojnih elemenata i ako je prvi element niza na adresi X, tada se adresa petog elementa dobija kao $X + broj\_memorijskih\_lokacija\_koje\_zauzima\_jedan\_ceo\_broj * 5$.

Ovo je ilustrovano na slici. Prema adresama skladištenja niza (22, 26, 2A, ...), vidimo da elementi niza zauzimaju 4 bajta. Adrese su zapisane u heksadecimalnom brojnom sistemu.

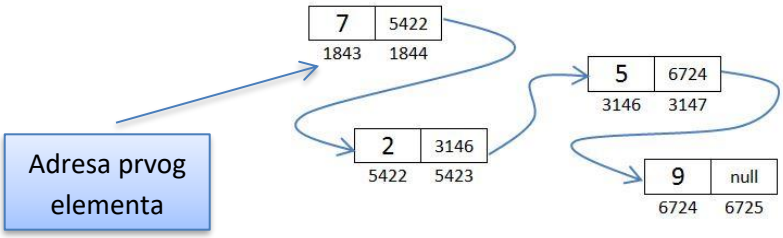


Sa druge strane, mana niza kao strukture podataka je slaba fleksibilnost. Obzirom da elementi moraju da zauzimaju uzastopne lokacije, nije jednostavno dodavanje novog elementa u niz. Ako se na memorijskog lokaciji koja se nalazi nakon poslednjeg elementa niza nalazi neki podatak, tada nije moguće jednostavno proširiti niz. Neophodno je zauzeti novi skup memorijskih lokacija dovoljan i za smeštanje novog elementa i kompletan niz prekopirati na te lokacije. Takođe, izbacivanje elementa niza podrazumeva da se svi elementi koji se nalaze nakon tog elementa moraju prebaciti za jedan element unazad ili se mora kreirati novi niz koji bi imao jedan element manje od originalnog niza i u koji bi se prekopirali svi elementi originalnog niza osim elementa koji se izbacuje.

Spregnuta lista

Druga linearna struktura podataka koje se često koristi je spregnuta lista. Ova struktura je fleksibilnija od niza, ali sa druge strane ima lošije performanse pri indeksiranju elementa.

Spregnuta lista elemente ne skladišti u uzastopnim memorijskim lokacijama. Redosled između elemenata je definisan time što svaki element skladišti i adresu na kojoj se nalazi njemu naredni element. Spregnuta lista je ilustrovana na sledećoj slici.

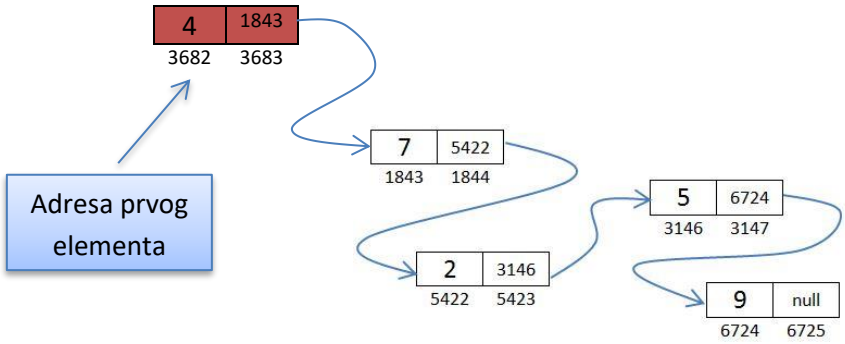


Kao što vidimo, svaki čvor u listi sadrži vrednost elementa, ali i adresu narednog elementa u listi. Ovakva organizacija ima za posledicu da nije moguće direktno adresirati proizvoljan element liste. Slično nizu, potrebno je uskladištiti adresu prvog elementa. Da bi se pristupilo proizvoljnom elementu, potrebno je iz prvog elementa preuzeti adresu drugog elementa, pa kada pristupimo tom elementu iz njega preuzeti adresu trećeg elementa i tako redom dok ne dođemo do željenog elementa. Dakle, lista ima lošije performanse pri indeksiranju proizvoljnog elementa liste u odnosu na niz.

Iz ovih razloga, niz pripada tipu struktura koje omogućuju proizvoljan pristup elementu (eng. *random access*). Sa druge strane, spregnuta lista omogućuje sekvencijalni pristup elementima, jer nije moguće pristupiti određenom elementu bez pristupa svim elementima koji mu prethode u sekvenci.

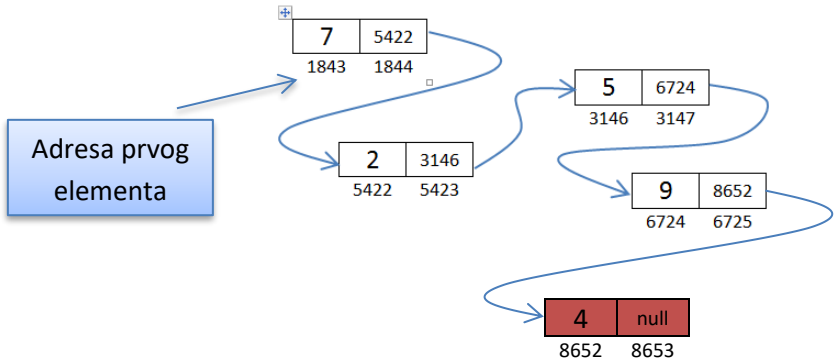
Kvalitet spregnute liste, kao strukture podataka, je njena fleksibilnost. Da podsetimo, to je bila značajna mana niza. Kod spregnute liste je puno jednostavnije umetanje novog elementa, kao i izbacivanje postojećeg. Obzirom da elementi ne moraju da zauzimaju uzastopne lokacije, novi element može biti kreiran na bilo kojoj lokaciji i uvezan u postojeću strukturu. Nije potrebno praviti potpuno novu strukturu sa kopiranjem originalnih podataka kao kod niza. Pri dodavanju novog elementa u listu postoje dve opcije kako se ova funkcionalnost može realizovati. Novi element može biti dodan na početak ili na kraj liste.

Naredna slika ilustruje dodavanje novog elementa sa vrednošću 4 na početak liste.



Kao što vidimo, novi element se upiše na bilo koju slobodnu lokaciju, a kao adresa njemu narednog elementa se postavlja adresa elementa koji je ranije bio prvi element u listi.

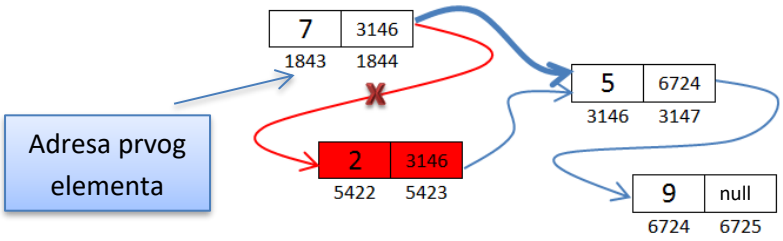
Naredna slika ilustruje dodavanje novog elementa na kraj liste.



I u ovom slučaju je potrebno novi element upisati na bilo koju slobodnu lokaciju. Element se uvezuje u listu tako što se elementu koji je do tada bio poslednji u listi postavi kao adresa narednog elementa adresa novog elementa. Novi element nema naredni element (adresa narednog mu je postavljena na null) obzirom da je on sada poslednji element.

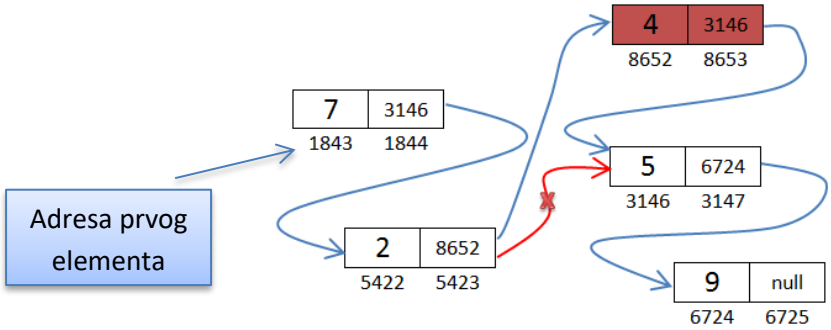
Spregnuta lista pokazuje svoju fleksibilnost i kod izbacivanja elementa iz liste. Dovoljno je izmeniti adresu narednog elementa u elementu koji prethodi obrisanom i element više neće biti uvezan u listu. Iako više nije uvezan u listu, izbačeni element će ostati u memoriji. U programskom jeziku Java, *garbage collector* će po potrebi osloboditi ovu memoriju, tako da nije potrebno vršiti nikakve

dodatne operacije pri izbacivanju elementa iz liste. Izbacivanje elementa iz spregnute liste je ilustrovano na sledećoj slici.



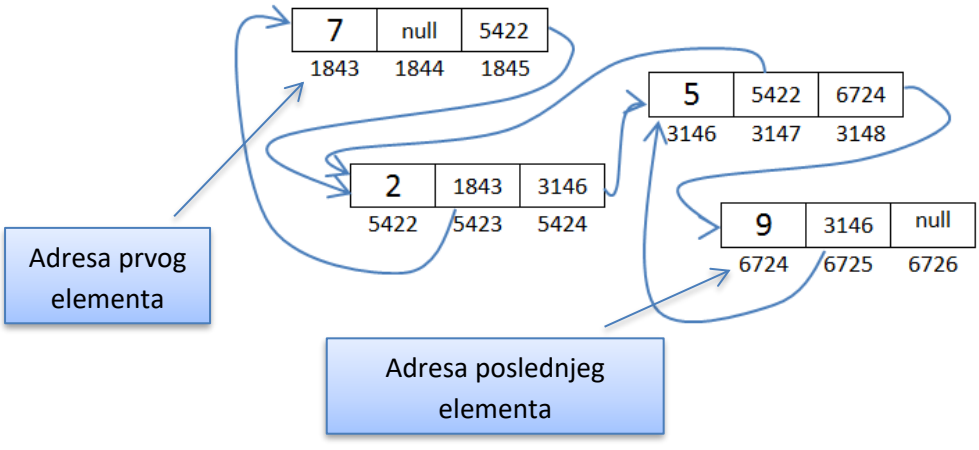
Ilustrovano je izbacivanje drugog elementa liste (elementa sa vrednošću 2). Kao što vidimo, u element koji mu prethodi, upisana je adresa trećeg elementa. Time element sa vrednošću 2 više nije ulančan u listu.

Moguće je i umetnuti novi element u listu, tj. dodati ga na prizvoljno mesto u listi. Ovo se vrši ažuriranjem adrese prethodnog elementa tako da pokazuje na umetnuti element. Ovo je ilustrovano na slici.



Vidimo da je novi element sa vrednošću 4 umetnut između elementa sa vrednošću 2 i elementa sa vrednošću 5.

Spregnuta lista kakva je prethodno opisana se naziva jednostruko spregnuta lista, jer svaki element sadrži samo jednu adresu (adresu narednog elementa). Druga varijanta realizacije spregnute liste je dvostruko spregnuta lista u kojoj svaki element skladišti adresu narednog elementa, ali i prethodnog elementa. Dvostruko spregnuta lista je ilustrovana na sledećoj slici.



Dakle, sada svaki element skladišti adresu i sebi prethodnog elementa. Vidimo da prvi element nema prethodnika, pa je na mestu tog podatke uskladištena vrednost null.

Skladištenje adrese i prethodnog i narednog elementa daje veću fleksibilnost pri prolasku kroz listu, obzirom da je moguće prolaziti kroz listu u oba smera. Ovo posebno dolazi do izražaja ako lista skladišti informaciju o adresama i prvog i poslednjeg elementa. Tada je moguće prolazak kroz listu krenuti od početka ili sa kraja liste. Sa druge strane, obzirom da elementi sadrže dve adrese, potrebno je više memorije za svaki element i pri operacijama nad listom neophodno je vršiti ažuriranje dve adrese, umesto jedne kao kod jednostruko spregnute liste.

Dodavanje elementa kod dvostruko spregnute liste je, kao i kod jednostruko spregnute liste, moguće realizovati uvezivanjem elementa na početak ili kraj liste ili na proizvoljno mesto unutar liste. U sva tri slučaja neophodno je ažurirati adrese u elementima koji prethode i slede elementu koji se ubacuje.

Slično, kod izbacivanja elementa iz dvostruko spregnute liste, potrebno je ažurirati adresu narednog elementa u elementu koji prethodi elementu koji se izbacuje. Takođe, neophodno je ažurirati adresu prethodnog elementa u elementu koji sledi elementu koji se izbacuje.

Moguće je dvostruko spregnutu listu implementirati i tako da poslednji element prethodi prvom, kao i da je prvi element naredni poslednjem elementu. Ovakva lista se naziva cirkularna lista. Takođe, i jednostruko spregnutu listu je moguće realizovati kao cirkularnu ako se postavi da je prvi element naredni poslednjem elementu.

Stek

Pomenuli smo da niz omogućuje direktan pristup proizvoljnom elementu (eng. *random access*), dok spregnuta lista predstavlja strukturu sa sekvencijalnim pristupom. Kod niza i spregnute liste bilo je moguće pristupiti bilo kojem elementu (bilo direktno, bilo prolaskom kroz elemente koji mu prethode). Takođe, sama struktura je omogućavala dodavanje ili izbacivanje elemenata sa proizvoljnog mesta u sekvenci (na početak, kraj ili negde unutar sekvence).

Sada ćemo predstaviti jedan specijalan tip sekvencijalnih struktura podataka. To su strukture podataka sa ograničenim pristupom elementima. Za razliku od ranije pomenutih struktura, kod struktura ovog tipa nije moguće pristupati proizvoljnom elementu liste, niti vršiti operacije dodavanja ili izmene na proizvoljno mesto u listi.

Prva struktura sa ograničenim pristupom elementima koju ćemo ovde predstaviti je stek. Stek je sekvencijalna struktura koja omogućuje sledeće operacije:

- Dodavanje elementa na početak sekvence
- Izbacivanje prvog elementa sekvence
- Preuzimanje prvog elementa sekvence (stek opciono sadrži ovu operaciju, zavisno od implementacije)

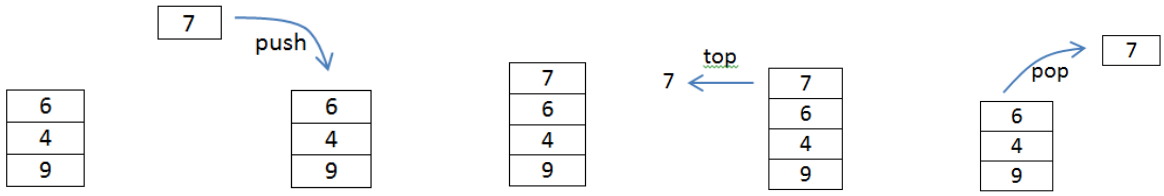
Dakle, vidimo da nije moguće pristupiti proizvoljnom elementu u steku i dodavanje je moguće samo na početak sekvence. Takođe, moguće je izbaciti samo prvi element.

Za strukturu koja ima ove osobine kažemo da je tipa LIFO (eng. *Last-in First-out*). Dakle, poslednji dodati element će prvi biti preuzet iz sekvence (jer se novi element dodaje na početak, a preuzimanje i izbacivanje pristupa elementu koji je na početku). Najčešće se stek ilustruje kao struktura koja vertikalno skladišti elemente, tako da je prvi element na vrhu. Tako koristimo pojam *vrh steka* za element koji je prvi u sekvenci.

Prethodno pomenute operacije dodavanja, preuzimanja i izbacivanja se najčešće nazivaju:

- push – dodavanje elementa na vrh steka
- pop – uklanjanje elementa koji je na vrhu steka
- top – preuzimanje elementa na vrhu steka (bez njegovog uklanjanja)

Stek i operacije koje omogućuje su ilustrovane na narednoj slici.



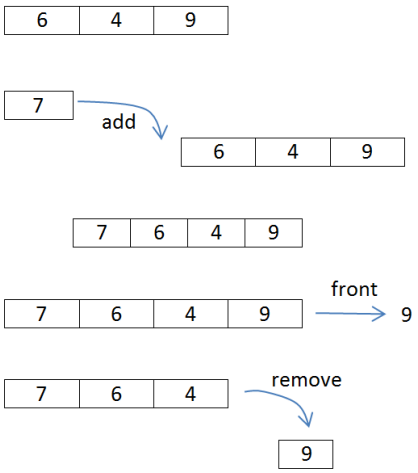
Zašto koristiti strukture sa ograničenim pristupom? Razlog je taj što pojednostavljuju skup mogućih operacija nad strukturom. U situacijama kada je kroz te operacije najefikasnije rešiti određeni problem, lakše je koristiti ovaj ograničeni skup operacija nego skup generalnih operacija nad sekvencom prilagođavati datom problemu. Na primer, stek se standardno koristi za implementaciju *undo* funkcionalnosti u tekstualnim editorima. Svaka izmena u tekstu se postavlja na stek da bi se pri operaciji *undo* uklanjala poslednje dodata izmena.

Red

Pored steka, ovde ćemo pomenuti još jednu često korišćenu sekvencijalnu strukturu sa ograničenim pristupom koja se naziva red (eng. *queue*). Za razliku od steka, red funkcioniše po FIFO (First-in First-out) principu. Dakle, element koji je prvi dodat u strukturu će prvi biti i preuzet iz nje. Možemo da kažemo da se elementi dodaju na kraj reda, a preuzimaju sa početka reda. Operacije koje red omogućuje su:

- dodavanje elementa na kraj reda (naziva se obično *add* ili *enqueue*)
- uklanjanje elementa sa početka reda (naziva se obično *remove* ili *dequeue*)
- preuzimanje elementa sa početka reda bez njegovog uklanjanja (red opciono sadrži ovu operaciju, zavisno od implementacije. Kada postoji, najčešće se naziva *front* ili *peek*)

Red i operacije koje omogućuje su ilustrovane na slici.



Ovde treba pomenuti i strukturu red sa dva kraja (*deque – double-ended queue*). Kod ove strukture, prema potrebi, elemente je moguće dodavati i uklanjati na oba kraja.

3. Unutrašnje klase

Iskoristićemo diskusiju o strukturama podataka da objasnimo još jedan konstrukt programskog jezika Java koji se često koristi upravo pri implementaciji struktura podataka. Reč je o unutrašnjim klasama. Naime, klasa u Javi, osim atributa i metoda koje smo do sada pominjali, može da sadrži i drugu klasu kao člana klase. Klase koje su definisane unutar neke druge klase zovemo unutrašnje klase. Pre ulaska u sintaksne detalje unutrašnjih klasa, obrazložićemo potrebu za njihovim postojanjem.

Posmatrajmo dvostruko spregnutu listu kao strukturu. Pri implementaciji ove strukture, listu ćemo predstaviti klasom. Objekat ove klase skladišti čvorove liste. Čvor takođe možemo modelovati posebnom klasom, koja kao atribute sadrži vrednost upisanu u čvor, kao i reference na prethodni, odnosno sledeći čvor. Klasa koja predstavlja čvor je kreirana u svrhu implementacije klase koja predstavlja listu. Iz tog razloga, ova klasa se definiše kao unutrašnja klasa, jer nije namenjena za korišćenje nezavisno od liste.

Pogledajmo sada kako se sintaksno definiše unutrašnja klasa, na primeru dvostruko spregnute liste i njenog čvora predstavljenog posebnom klasom. Čvor skladišti celobrojne vrednosti.

```
class DoubleLinkedList {
	private ListItem head;
	...
	private class ListItem {
		private int content;
		private ListItem prev;
		private ListItem next;
		public ListItem(int content, ListItem prev, ListItem next) {
			this.content = content;
			this.prev = prev;
			this.next = next;
		}
	}
}
```

Vidimo da unutrašnja klasa ima istu strukturu kao i standardna klasa, samo što se definiše unutar druge klase. Vidimo da je modifikator pristupa za datu unutrašnju klasu *private*. Ovako definisana klasa nije dostupna spolja, već je vidljiva samo iz klase u okviru koje je definisana. Unutrašnja klasa se može definisati i kao *public* ili *protected*. U tom slučaju je spolja dostupna i pristupa joj se preko objekta klase u kojoj je definisana. Na primer, ako imamo klase Spoljasnja i Unutrasnja, pri čemu je klasa Unutrasnja definisana kao javna unutrašnja klasa u okviru klase Spoljasnja, tada bi se klasi Unutrasnja moglo pristupi na sledeći način.

```
Spoljasnja s = new Spoljasnja();
Spoljasna.Unutrasnja u = s.new Unutrasnja();
```