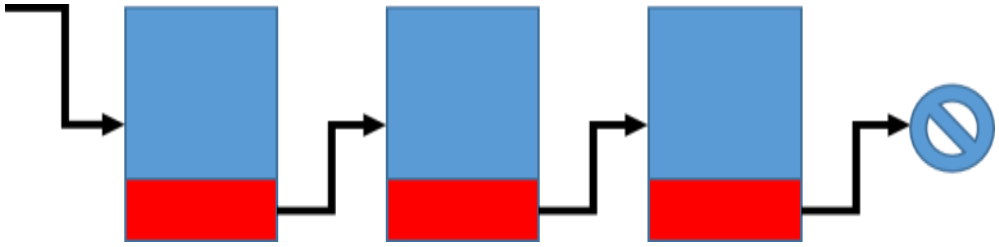
1. Dinamički vezane liste

U ovom poglavlju ćemo vidjeti alternativan način za implementaciju apstraktnog tipa liste. Do sada je lista bila implementirana kao niz i to je generalno najčešći slučaj. Svejedno, ATP lista se može implementirati i puno fleksibilnije u obliku lanca elemenata. Ova implementacija u C jeziku se oslanja na pokazivače i potrebno je razumjeti memorijski model koji se nalazi u pozadini programskih jezika. Sama ideja je jednostavna, ali implementacija može biti zbunjujuća na prvi pogled. Svaki element u listi će imati dodatak koji će određivati gdje se nalazi idući element. Cijela lista će biti predstavljena prvim elementom ili glavom liste. Posljedica ovakve implementacije, u odnosu na nizove, je da više ne možemo direktno pristupati bilo kojem elementu u listi već moramo „šetati“ od glave liste. Vidjeti ćemo i varijante koje omogućuju da se ublaže neka ograničenja osnovne „jednostruko vezane“ liste. Ovdje će se termin „lista“ koristiti kao skraćeni termin za „dinamičku jednostruko vezanu listu“.



Osnovne operacije nad listama

Listu ćemo u C programskom jeziku definirati kao strukturu sastavljenu od dijela koji sadrži informacije samog elementa (npr. broj) i dijela koji služi za povezivanje sa ostatkom liste. U C-u će se taj dio realizirati kao jednostavan pokazivač na idući element liste.

```
typedef struct _Element {
      int broj; // stvarni sadržaj elementa
      struct _Element *next; // pokazivač na idući element liste
} Element;

void main() {
      Element * lista = NULL; // prazna lista

}
```

Prazna lista će biti pokazivač na NULL vrijednost koja će označavati i kraj liste. Kada lista ima jedan ili više elemenata, zadnji element liste će uvijek pokazivati na NULL koji će označavati kraj liste. Za sve iduće primjere ćemo koristiti strukturu *Element*, ali sve vezano uz samu „infrastrukturu“ liste će vrijediti i za elemente koji sadrže drugačiji sadržaj elemenata. Ubuduće ćemo sve elemente alocirati sa malloc(), ali za početak možemo sastaviti listu od nekoliko elemenata stvorenih na programskom stogu.

```
void main() {
	Element *lista = NULL; // prazna lista
	Element a, b, c;
	a.broj = 1;
	b.broj = 2;
	c.broj = 3;
	c.next = NULL;
	b.next = &c;
	a.next = &b;
	lista = &a; // lista -> 1 -> 2 -> 3 -> NULL
}
```

Članovi strukture *next* će posjedovati adresu idućeg elementa i krenuvši od pokazivača *lista* možemo doći do bilo kojeg elementa. Naravno, sve operacije nad listama ćemo kasnije organizirati u prikladne funkcije. Za početak možemo napraviti funkciju koja će ispisati navedenu listu.

```
void ispisi_listu(Element *lista) {
	Element *tmp = lista;
	while (tmp != NULL) {
		printf("%d ", tmp->broj);
		tmp = tmp->next;
	}
}
```

Ova funkcija pokazuje osnovni princip šetanja po listi i možemo da usporediti sa šetanjem po nizu gdje ne znamo dužinu niza, ali znamo kako izgleda kraj niza (poput string terminatora). Pokazivač *tmp* nam služi umjesto brojača i pokazuje na trenutni element liste tijekom while petlje. Na početku ga pozicioniramo na prvi element. Kada *tmp* postane NULL, znači da smo došli do kraja liste. U samoj petlji imamo ispis sadržaja (*broj*) trenutnog elementa i tipičnu naredbu za pomicanje pokazivača na idući element liste. Ta naredba u pokazivač *tmp* smješta adresu idućeg elementa koji se nalazi u članu *next* trenutnog elementa. To je ekvivalent „i++“ naredbe kod nizova. Samu listu prosljeđujemo funkciji kao običan pokazivač odnosno adresu prvog elementa liste. U ovom slučaju

smo mogli koristiti pokazivač *lista* kao šetača po lista, ali to je samo zato što nam početak liste nije potreban nakon što ga ispišemo.

U takvu (jednostruko vezanu) listu je najlakše dodavati elemente „u glavu“ odnosno na početak liste jer imamo direktan pristup preko pokazivača na prvi element. Možemo napraviti funkciju koja će kreirati i dodati novi broj kao element liste.

```
Element* dodaj_u_glavu(Element *lista, int broj) {
	Element *novi = (Element*) malloc(sizeof(Element));
	novi->broj = broj;
	novi->next = lista;
	lista = novi;
	return lista;
}
```

Prva linija funkcije kreira novi element pomoću malloc() funkcije. U sadržaj kopiramo primljeni broj, a preko *next* člana povezujemo element sa ostatkom liste. Iduća naredba postavlja pokazivač *lista* na novi element i time je sada lista uvećana za novi element na početku liste. Da bi se taj efekt osjetio i izvan funkcije moramo nekako vratiti novu listu. Bez vraćanja u zadnjoj liniji (i hvatanja vraćene vrijednosti pri pozivu funkcije), promijenili bi parametar *lista*, ali to ne bi imalo nikakvog efekta na originalnom pokazivaču. Isto bi mogli postići i sa dvostrukim pokazivačem odnosno pristupajući originalnom pokazivaču.

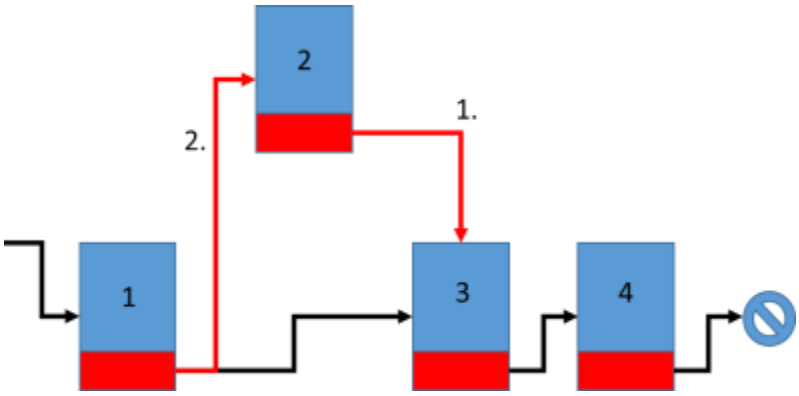
```
void dodaj_u_glavu(Element **lista, int broj) {
	Element *novi = (Element*)malloc(sizeof(Element));
	novi->broj = broj;
	novi->next = *lista;
	*lista = novi;
}
```

Malo kompliciranije je dodati element na kraj jednostruko vezane liste. Za to je potrebno da prošetamo do zadnjeg elementa liste i obavimo spajanje na novi element. Uz to je potrebno posebno obraditi slučaj kada je primljena lista prazna.

```
Element* dodaj_na_kraj(Element *lista, int broj) {
	Element *novi = (Element*)malloc(sizeof(Element));
	novi->broj = broj;
	novi->next = NULL;
	if (lista == NULL) // ako je lista prazna
		return novi;
	// šetamo do zadnjeg elementa
	Element *tmp = lista;
	while (tmp->next != NULL) {
		tmp = tmp->next;
	}
	// spajamo kraj s novim elementom
	tmp->next = novi;
	return lista;
}
```

Bitno je primijetiti malu promjenu u uvjetu while petlje. Ideja je da provjeravamo *next* član trenutnog elementa tako da petlja završi kada smo se pozicionirali na zadnjem elemente, a ne na NULL vrijednosti.

Sličan slučaj je kada novi element želimo dodati negdje po sredini liste. I dalje moramo šetati po listi, ali sada se možemo zaustaviti bilo gdje. Na primjer, možemo napisati funkciju koja će dodati novi broj u listu koja je već sortirana po veličini.



```
Element* dodaj_po_redu(Element *lista, int broj) {
	Element *novi = (Element*)malloc(sizeof(Element));
	novi->broj = broj;
	novi->next = NULL;
	if (lista == NULL) // ako je lista prazna
		return novi;
	if (novi->broj < lista->broj) {
		// dodajemo na prvo mjesto
		novi->next = lista;
		return novi;
	}
	// šetamo do elementa koji je veći od broja
	Element *tmp = lista;
	while (tmp->next != NULL) {
		if (novi->broj < tmp->next->broj) {
			novi->next = tmp->next;
			tmp->next = novi;
			return lista;
		}
		tmp = tmp->next;
	}
	// dodajemo na kraj liste
	tmp->next = novi;
	return lista;
}
```

Opet, logika se može usporediti sa dodavanjem u sortirani niz, ali nema „guranja“ preostalih elemenata da bi napravili mjesta za novi element. To i je najveća prednost dinamički vezanih lista – efikasnost kod dodavanja i brisanja elemenata.

Operacija brisanja elementa je opet najjednostavnija na početku liste. Pod pretpostavkom da su elementi stvoreni pomoću malloc() poziva, moramo ih brisati oslobađajući memoriju sa free() pozivom. Uz to, treba paziti da se ne izgubi dio liste i prethodno treba „premostiti“ element koji brišemo.

```
Element *brisi_glavu(Element *lista) {
	if (lista == NULL)
		return NULL;
	Element *brisi = lista;
	lista = lista->next;
	free(brisi);
	return lista;
}
```

Brisanje određenog elementa je jednako kao brisanje kraja liste, jedino što ćemo element koji želimo brisati prepoznati po tome što njegov član *next* pokazuje na NULL.

```
Element *brisi_element(Element *lista, int broj) {
	if (lista == NULL)
		return NULL;
	if (lista->broj == broj) {
		// brisanje prvog elementa
		Element *brisi = lista;
		lista = lista->next;
		free(brisi);
		return lista;
	}
	Element *tmp = lista;
	while (tmp->next != NULL) {
		if (tmp->next->broj == broj) {
			Element *brisi = tmp->next;
			tmp->next = tmp->next->next;
			free(brisi);
			return lista;
		}
		tmp = tmp->next;
	}
	// ako element nije pronađen u listi
	return lista;
}
```

Generalno, ovakve funkcije je možda najpraktičnije pisati idućim redoslijedom: generalni slučaj za operacije negdje po sredini liste, a zatim obraditi posebne slučajeve na početku ili kraju liste.

Složenije operacije nad listama

Ovdje ćemo pokazati i neke složenije operacije nad listama. Takve operacije obično rade sa svim elementima liste ili sa više lista, ali njihova implementacija može biti i jednostavnija od osnovnih operacija koje smo vidjeli.

Za početak, često želimo osloboditi cijelu memoriju koju lista zauzima brisanjem cijele liste. To je najlakše napraviti uzastopnim brisanjem glave liste dok ne dođemo do NULL vrijednosti.

```
void brisi_listu(Element *lista) {
	while (lista != NULL) {
		Element *brisi = lista;
		lista = lista->next;
		free(brisi);
	}
}
```

Pri pozivanju ove funkcije treba paziti da ona neće postaviti originalni pokazivač na NULL, pa je to potrebno napraviti nakon poziva funkcije da se lista označi kao prazna.

Za spajanje dvije liste, dovoljno je prošetati do zadnjeg elementa prve liste i preusmjeriti njegov *next* na početak druge liste. Poseban slučaj je da je prva lista prazna.

```
Element* brisi_listu(Element *lista1, Element *lista2) {
	if (lista1 == NULL)
		return lista2;
	Element *tmp = lista1;
	while (tmp->next != NULL) {
		tmp = tmp->next;
	}
	tmp->next = lista2;
	return lista1;
}
```

Okretanje liste je jednostavno jer sve operacije radimo na glavama originalne i (nove) okrenute liste. Nema potrebe da išta alociramo ili oslobađamo, ali pri tome se originalna lista uništava.

```
Element* okreni_listu(Element *lista) {
	Element *okrenuta = NULL;
	while (lista != NULL) {
		Element *tmp = lista;
		lista = lista->next;
		tmp->next = okrenuta;
		okrenuta = tmp;
	}
	return okrenuta;
}
```

Napraviti kopiju liste je složenije jer moramo kreirati i dodavati elemente na kraj kopije. Najefikasnije je uvijek držati pokazivač na zadnji element kopije. Ova funkcija ovisi i o sadržaju elemenata jer ih moramo kopirati. Ovdje je prikazana verzija za listu brojeva, a generalnija verzija bi se eventualno mogla oslanjati na memcpy() ili neki drugi način kopiranja sadržaja.

```
Element* kopiraj_listu(Element *lista) {
	Element *kopija = NULL;
	Element *kraj = NULL;
	while (lista != NULL) {
		Element *novi = (Element*) malloc(sizeof(Element));
		novi->broj = lista->broj;
		novi->next = NULL;
		// samo zbog prvog elementa
		if (kraj != NULL)
			kraj->next = novi;
		else
			kopija = novi;
		kraj = novi;
		lista = lista->next;
	}
	return kopija;
}
```

Sortiranje liste se, uz manje izmjene, može izvesti pomoću viđenih algoritama u prethodnom poglavlju. Ovdje ćemo samo opisati ideju i složenost za svaki od algoritama sortiranja za dinamički vezane liste.

Selectionsort bi jednostavno gradio novu listu ubacujući u glavu najveći element iz originalne liste. Pri tome bi taj element premostili i time ga izbacili iz originalne liste. Složenost bi se mogla izračunati kao O(n) traženja najvećeg elementa u originalnoj list koja se svaki put smanjuje za jedan element. Svako traženje bi nas koštalo O(n), a ubacivanje u glavu nove liste bi koštalo O(1). Na kraju dobijemo kvadratnu složenost $O(n^2)$ kao i u algoritmu za nizove. Insertionsort bi uzimao glavu originalne liste i ubacivao je po redu u novu listu. Opet, traženje mjesta u novoj listi bi se ponavljalo O(n) puta, a koštalo bi nas proporcionalno dužini sortirane liste koja bi se uvećavala za jedan svaki put. Opet, ukupna složenost je $O(n^2)$.

Quicksort bi se mogao izvesti birajući pivot i dijeleći listu na „manji“ i „veći“ dio. Svejedno, to bi bilo dosta neefikasno u usporedbi s mergesort algoritmom jer je glavna mana mergesort algoritma bila dodatna O(n) memorija potrebna za sortiranje. U slučaju mergesort algoritma za dinamički vezane liste, potrebna je samo O(logN) dodatna memorija (zbog rekurzivnih poziva), a mergesort nam ujedno rješava i problem najgore složenosti quicksort algoritma. Mergesort bi jednostavno dijelio listu na pola tako da svaki drugi element prebacujemo u drugu listu. Tako dobijemo dvije liste koje dalje dijelimo rekurzivno. Na povratku iz rekurzija možemo prilagođenom merge() funkcijom spojiti dvije liste i dobiti sortiranu listu. Složenost je na kraju i dalje O(NlogN) za bilo koju listu.

Usporedba lista i nizova

Nizovi imaju više prednosti u odnosu na dinamički vezane liste i u praksi su najčešće korišteni. Ovdje ćemo spomenuti samo najvažnije razlike.

Nizovi omogućuju direktan pristup bilo kojem elementu. Liste omogućuju direktan pristup samo prvom elementu.

Nizovi su lakši za korištenje ako nema potrebe za čestim uklanjanjem ili dodavanjem elemenata po sredini ili na početku niza. Liste su fleksibilnije za izmjenu, ali svejedno najčešće moramo „došetati“ do elementa koji mijenjamo.

Šetanje po nizu je efikasnije u praksi jer će biti pakiran kao blok memorije i procesorski *cache* (i druge memorije) će lakše predvidjeti i učitati potrebne podatke iz glavne memorije (RAM). Liste će uglavnom biti „razbacane“ po memoriji.

U konačnici, dinamički vezane liste su zgodne samo kada imamo puno izmjena na početku liste ili kada nam je prebacivanje elemenata naročito skupo (npr. kod „liste“ blokova na disku). Svejedno, princip dinamičkog vezivanja elemenata nam omogućava realizaciju složenijih struktura podataka koje ćemo vidjeti kasnije, poput stabala.

Varijante

Da bi se ublažili neki nedostaci jednostruko vezanih lista, osmišljene su i neke varijante koje u nekim slučajevima mogu ubrzati ili olakšati rad sa listama.

Do sada smo koristili liste koje nemaju poseban element kao „glavu“ liste. Postoje još dvije mogućnosti: „dummy“ glava i posebna glava.

U slučaju dummy glave, lista uvjek ima najmanje jedan element u koji ne spremamo nikakav sadržaj. Taj element bi imao isti tip kao i ostali elementi liste, a prava lista bi počinjala od njegovog *next* člana koji bi za praznu listu pokazivao na NULL. Ideja je da se izbjegne potreba za povratnom vrijednošću ili dvostrukim pokazivačima u funkcijama koje mijenjaju listu. Takva glava ne mora nužno biti alocirana na hrpi nego može biti jednostavno deklarirana na programskom stogu. Svi primjeri koje smo vidjeli bi trebali to uzeti u obzir i raditi od „drugog“ elementa.

Slično je i sa listom sa posebnom glavom gdje bi prvi element bio posebni tip (strukture). Taj element bi eventualno sadržao nekakve dodatne informacije uz pokazivač koji pokazuje na početak prave liste. Osim što nam olakšava rad sa listom (kao u slučaju dummy glave), taj posebni tip za glavu liste može sadržati i neke druge pokazivače poput pokazivača na zadnji element liste.

Jedna od glavnih mana jednostruko vezanih lista je da ih možemo prolaziti samo u jednom smjeru, a ne možemo se jednostavno vratiti korak unazad. Dvostruko vezane liste rješavaju taj problem nauštrb dodatne memorije. Svaki element će, osim pokazivača na idući element *next*, sadržati i pokazivač na prethodni element - *pred*. Ovakva implementacija ima složenije i sporije operacije na listi jer je potrebno manipulirati sa dvostruko većim brojem pokazivača.

Postoji i složenija varijanta sortirane dinamički vezane liste koja pokušava smanjiti vrijeme traženja elementa u listi – *skip* liste. Ideja je da svaki element liste sadrži dodatne pokazivače koji pokazuju na neke elemente u ostatku liste i efektivno „preskaču“ veliki dio elemenata liste. Takva lista se obično vizualizira u obliku razina gdje najniža razina je obična jednostruko vezana lista svih elemenata, a svaka viša razina preskače sve veći broj elemenata. Kako je lista sortirana, pomoću preskakanja elemenata (kroz više razine) i usporedbom možemo postići složenost pretrage sličnu binarnoj pretrazi O(logN).

Zadaci

Napisati verziju funkcije bez povratne vrijednosti (sa dvostrukim pokazivačem):

dodaj_na_kraj()

dodaj_po_redu()

izbrisi_glavu()

izbrisi_element()

Napisati rekurzivnu verziju funkcije za brisanje cijele liste. Odrediti memorijsku složenost ovakve funkcije.

Implementirati insertionsort, selectionsort i mergesort algoritme za dinamički vezane liste.