Programiranje II, tok 101, šk. 2014/15. g.

dr Gordana Pavlović-Lažetić

1

Uvod

Program predmeta Programiranje II uključuje sledeće teme:

1. Operatori nad bitovima
2. Složenost algoritama; asimptotske notacije $O(n)$, $\Omega(n)$, $\Theta(n)$
3. Elementarne metode sortiranja: sortiranje izborom najvećeg, najmanjeg elementa, sortiranje umatenjem; složenost
4. Rekurzija; stek memorija; principi konstrukcije; primeri
5. Rekurzivne funkcije za binarno pretraživanje i sortiranje (brzi sort – quicksort, sortiranje spajanjem – merge sort); složenost
6. Repna rekurzija - eliminacija
7. Pokazivači (aritmetika, saglasnost, void *)
8. Funkcije za alokaciju memorije - malloc, calloc; heap memorija
9. Pokazivači na funkcije; funkcije - argumenti funkcija; standardne funkcije lfind, bsearch i qsort;
10. Dinamičke strukture; liste, red, stek
11. Dinamičke strukture – drvo, binarno drvo, obilazak; uređeno binarno drvo
12. Funkcije pretraživanja (inicijalizacije, umetanja) dinamičkih struktura
13. Numerički-nenumerički algoritmi – polinomi, sravnjivanje niski, geometrijski algoritmi, algoritmi nad grafovima
14. Algoritmi traganje unazad (backtracking); primeri
15. Formalna definicija pojma algoritam; algoritamski nerešivi problemi

16. Klasifikacija jezika; programske paradigme

Pored ovih tema, u materijal su uključene i teme koje u ovoj školskoj godini ne pripadaju ovom predmetu, a čine celinu sa izloženim, i to:

1. Datoteke, FILE struktura; funkcije fopen, freopen; argumenti komandne linije; funkcije direktnog pristupa datotekama fseek, ftell
2. Promenljivi broj argumenata funkcije
3. Dinamičke strukture – grafovi, reprezentacija
4. Direktno (heš) pretraživanje

2

Datoteke – ulaz i izlaz

Datoteka je niz znakova (bajtova) kome je pridruženo ime, i koji se može interpretirati na razne načine – npr. kao tekst, kao program na programskom jeziku, kao program preveden na mašinski jezik, kao slika, crtež, zvuk, itd. Na sadržaj datoteke tj. način na koji ga treba interpretirati, ukazuje ekstenzija uz ime datoteke (npr. c, exe, txt, pas, obj, exe, bmp, itd). (Pojam spoljašnje datoteke se bitno razlikuje od pojma datoteke kao strukture podataka, u nekim programskim jezicima, npr. Pascal-u).

U programskom jeziku C datoteka je pre svega vezana za ulaz i izlaz podataka. Mogućnosti ulaza i izlaza nisu sastavni deo samog jezika C, ali su neophodne za komunikaciju programa sa okruženjem i mogu biti znatno kompleksnije od onih koje smo neformalno koristili do sada (getchar, putchar, printf, scanf). Ovaj aspekt jezika podržan je funkcijama standardne biblioteke, čija su svojstva precizirana u preko deset standardnih zaglavlja – neka od njih su već korišćena – <stdio.h>, <string.h>, <ctype.h>, itd.

2.1 Standardni ulaz i izlaz

Standardna biblioteka C-a uključuje jednostavni model tekstualnog ulaza i izlaza. Ulazni tekst se sastoji od niza linija, svaka linija se završava markerom kraja reda (jednim ili više karaktera, u zavisnosti od operativnog sistema – npr. karakter <lf> ili par karaktera <cr>, <lf>).

Najjednostavniji mehanizam za čitanje jednog karaktera sa standardnog ulaza (obično tastature), kako je već rečeno, predstavlja funkcija

int getchar(void)

koja vraća sledeći ulazni karakter, ili EOF kada se dođe do kraja. Simbolička konstanta EOF je definisana u zaglavlju <stdio.h>, i obično ima vrednost -1.

U mnogim okruženjima tastatura se može zameniti datotekom, korišćenjem *konvencije preusmerenja*, < Ako program prog koristi funkciju getchar, onda se on može izvršiti komandom

prog <indat
pri čemu se karakteri čitaju iz datoteke indat umesto sa tastature.

Najjednostavniji mehanizam za izdavanje jednog karaktera na standardni izlaz (obično ekran), kako je već ranije rečeno, predstavlja funkcija

int putchar(int)

koja, pozvana sa putchar(c) izdaje karakter c na standardni izlaz i vraća karakter c ili EOF ako dođe do greške. Opet se, umesto standardnog izlaza, može koristiti i datoteka (npr. outdat), preusmerenjem oblika

prog >outdat

Izveštaj o izvršavanju programa i eventualnim greškama ispisuje se obično na standardni izlaz, ali ga je moguće preusmeriti na neku datoteku (npr. greska) operatorom preusmerenja oblika

prog 2>greska

Sva tri preusmerenja na specifične datoteke postižu se sledećim zapisom:

prog <indat >outdat 2>greska

Svaka izvorna datoteka koja koristi funkcije ulazno/izlazne biblioteke, mora da uključi zaglavlje <stdio.h>.

2.1.1 Formatirani izlaz

Kao što je već pomenuto u delu "Pregled programskog jezika C", funkcija formatiranog izlaza na standardni izlaz ima deklaraciju oblika

int printf(char *format, arg_1, arg_2, ...)

Karakteristični za string format, pod čijom kontrolom se argumenti konvertuju i izdaju na standardni izlaz, jesu objekti za specifikaciju konverzije koji govore kojim tipom interpretirati i u koji oblik konvertovati svaki od argumenata. Ovi objekti počinju znakom % i završavaju se karakterom konverzije – nekim od karaktera "d", "i" (za izdavanje celog broja), "c" za izdavanje jednog karaktera, "s" za izdavanje stringa (niske karaktera), "f" za izdavanje realnog broja, itd.

Između znaka % i karaktera konverzije, objekat za specifikaciju konverzije može da uključi i druge karaktere specifičnog značenja:

- znak minus za levo poravnanje
- broj koji određuje minimalnu širinu polja
- tačku, koja razdvaja širinu polja od broja koji označava preciznost
- broj koji označava preciznost – maksimalni broj karaktera koji će se izdati, ili broj cifara iza decimalne tačke broja u pokretnom zarezu, ili najmanji broj cifara celog broja
- karakter h ili l za kratki odnosno dugi ceo broj.

Slično funkciji printf, funkcija
int sprintf(char *string, char *format, arg_1, arg_2, ...)
konvertuje i formatira argumente prema stringu format, ali odgovarajuće karaktere ne izdaje na standardni izlaz već ih upisuje u string.

2.1.2 Formatirani ulaz

Kao što je već pomenuto u delu "Pregled programskog jezika C", funkcija scanf jeste analogon funkcije printf za formatirani unos:
int scanf(char *format, arg_1, arg_2, ...)
Ona uzima nisku karaktera sa standardnog ulaza, konvertuje je u vrednosti svojih argumenata i dodeljuje ih promenljivim na koje pokazuju argumenti arg_1, arg_2, ... Dakle, svi argumenti funkcije scanf jesu pokazivači.

Slično funkciji sprintf, funkcija
int sscanf(char *string, char *format, arg_1, arg_2, ...)
konvertuje nisku karaktera u vrednosti svojih argumenata, ali nisku karaktera ne uzima sa standardnog ulaza već iz niske string.

String format može da sadrži:

- beline ili tabulatore koji se ignorišu
- obične karaktere (isključujući %) koji se očekuju na ulazu
- objekte za specifikaciju konverzije koji se sastoje od znaka % i karaktera konverzije (sa sličnim značenjem kao kod printf), i, između njih, opcionog karaktera * kojim se "preskače" ulazno polje, bez dodele vrednosti promenljivoj, opcionog broja koji određuje maksimalnu dužinu polja, i opcione oznake kratkog odnosno dugog celog broja.

2.2 Pristup datotekama

Sva komunikacija programa sa okruženjem do sada je podrazumevala standardni ulaz odnosno standardni izlaz, čiju definiciju program automatski dobija od operativnog sistema.

Postoji potreba – i mogućnost – da program pristupi imenovanoj datoteci koja nije povezana sa programom, tj. da se povežu spoljašnje ime (datoteke) i programski iskaz kojim se čitaju podaci iz te datoteke. To se postiže *otvaranjem* datoteke bibliotečkom funkcijom fopen, koja uzima spoljašnje ime datoteke (npr program.c) i vraća pokazivač, tzv. *pokazivač datoteke*, koji pokazuje na strukturu sa informacijama o toj datoteci. Definicija tipa te strukture zove se FILE i nalazi se u standardnom zaglavlju <stdio.h>. Deklaracija funkcije fopen ima oblik

FILE *fopen(const char *ime, const char *nacin);

gde ime označava string koji sadrži spoljašnje ime datoteke, nacin je nameravani način korišćenja datoteke – "r" za čitanje, "w" za upis", "a" za dodavanje.

Da bi datoteka mogla da se koristi, funkcija fopen mora da se pozove sa odgovarajućim stvarnim parametrima (argumentima), a njena vrednost da se dodeli pokazivaču na strukturu tipa FILE koji će se nadalje koristiti za čitanje iz tj. upis u datoteku:

FILE *fp;

fp = fopen(ime, nacin);

Otvaranje (postojeće) datoteke za upis (odnosno dodavanje) briše (odnosno čuva) prethodni sadržaj te datoteke, dok otvaranje nepostojeće datoteke za upis ili dodavanje kreira tu datoteku. Pokušaj otvaranja nepostojeće datoteke za čitanje proizvodi grešku (fopen vraća NULL).

Datoteka otvorena na prethodni način tretira se kao tekstualna datoteka, tj. datoteka koja sadrži redove teksta. Na kraju svakog reda se automatski upisuje marker kraja reda (jedan ili više karaktera, u zavisnosti od operativnog sistema - npr. <lf> (ASCII kod 10) ili <cr><lf> (ASCII kodovi 13 10). Pri čitanju iz tekstualne datoteke u string, marker kraja reda konvertuje se u nula-karakter (ASCII kod 0) kojim se završavaju stringovi u C-u.

"Način" u deklaraciji funkcije fopen može da ukaže i na to da datoteku otvaramo na "binarni" način, tj. da će se otvorena datoteka tretirati kao binarna datoteka. Tada "način" može da bude "rb" za čitanje, "wb" za upis i "ab" za dodavanje.

Pri čitanju iz binarne datoteke, ako se u njoj naiđe na marker kraja reda, karakteri kojima je on predstavljen (u zavisnosti od operativnog sistema) učitavaju se kao takvi u string i tretiraju se kao i svaki drugi karakter. Pri upisu u binarnu datoteku nema automatskog upisa markera kraja reda, već se on upisuje samo ako ga korisnik eksplicitno unese – i to onako kako ga unese, nezavisno od operativnog sistema.

Binarna datoteka može da sadrži tekst, ali taj tekst nije razbijen u redove markerima kraja reda. Binarna datoteka može da sadrži i netekstualnu informaciju. Program za čitanje binarne datoteke treba da je konvertuje u nešto smisleno – sliku, zvuk, tekst i sl.

Postoji više načina za čitanje iz ili upis u datoteku otvorenu za čitanje odnosno upis. Najjednostavniji je par funkcija

int fgetc(FILE *fp)
int fputc(int c, FILE *fp)

odnosno

int getc(FILE *fp)
int putc(int c, FILE *fp).

kojima se, slično funkcijama getchar, putchar za standardni ulaz odnosno izlaz, čita sledeći karakter datoteke na čiju strukturu pokazuje pokazivač fp, odnosno upisuje karakter c. Funkcije getc i putc mogu da budu realizovane i kao makro definicije.

Za čitanje i upis niske karaktera koriste se funkcije
char *fgets(char *s, int maxduzina, FILE *fp),

koja učitava sledeću ulaznu liniju (uključujući karakter za novi red), ili njen deo duzine maxduzina, ali najviše maxduzina karaktera, iz datoteke fp otvorene za čitanje u nisku s, odnosno

int fputs(const char *s, FILE *fp),

koja upisuje nisku s u datoteku fp otvorenu za upis.

Pri startovanju programa (izvršavanju), operativni sistem automatski otvara tri standardne datoteke (definisane u zaglavlju <stdio.h>): stdin (standardni ulaz, obično tastatura), stdout, stderr (standardni izlaz i standardna datoteka za izveštavanje o greškama – obično ekran). Stdin, stdout i stderr su zapravo pokazivači odgovarajućih datoteka, dakle objekti tipa FILE *, ali su to konstantni objekti tog tipa (konstante) i ne mogu im se dodeljivati (menjati) vrednosti.

Sada se funkcije getchar, putchar mogu definisati kao odgovarajuće funkcije getc, putc nad odgovarajućom standardnom datotekom (stdin odnosno stdout):

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
```

Za formatirani ulaz/izlaz iz/u datoteku, mogu se koristiti funkcije analogne funkcijama scanf, printf, ali za datoteku čiji je pokazivač datoteke fp:

int fscanf(FILE *fp, char *format, arg_1, arg_2, ...)
int fprintf(FILE *fp, char *format, arg_1, arg_2, ...)

Najzad, funkcija
int fclose(FILE *fp)
raskida vezu između pokazivača datoteke i spoljašnjeg imena, koja je bila uspostavljena pozivom funkcije fopen. Pošto postoji ograničenje na broj istovremeno otvorenih datoteka, dobro je zatvoriti svaku datoteku čim prestane njeno korišćnje.

Na primer, funkcija koja kopira sadržaj datoteke sa pokazivačem ulazp u datoteku sa pokazivačem izlazp ima sledeću definiciju:

```
/*filecopy: kopira datoteku ulazp u datoteku izlazp */
void filecopy(FILE *ulazp, FILE *izlazp)
{
    int c;
    while ((c=getc(ulazp)) != EOF)
        putc(c, izlazp);
}
```

a program koji poziva ovu funkciju:

```
#include <stdio.h>
void filecopy(FILE *, FILE *);
/*kopiranje: kopira datoteku a.c u datoteku b.c */
int main()
{
    FILE *ulazp, *izlazp;
```

```
    if ((ulazp=fopen("a.c", "r"))==NULL) {
        printf("ne moze da se otvori datoteka a.c\n");
        return 1;
    }
    else {
        if ((izlazp=fopen("b.c", "w"))==NULL) {
            printf("ne moze da se otvori datoteka b.c\n");
            return 1;
        }
        else {
            filecopy(ulazp, izlazp);
            fclose(izlazp);
            izlazp=fopen("b.c", "r");
            filecopy(izlazp, stdout);
            fclose(izlazp);
        }
        fclose(ulazp);
    }
}
```

2.3 Alternativno preusmeravanje - funkcija `freopen()`

Pored funkcije `fopen()` za otvaranje datoteke može da se koristi i funkcija `freopen()`. Kao i druge funkcije ulaza i izlaza, definisana je u standardnom zaglavlju <stdio.h> Njeno zaglavlje je oblika

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

Funkcija `freopen()` otvara datoteku sa imenom `filename` (npr. "ualz.txt") na navedeni nacin (`mode`, npr. "r") i pridužuje joj pokazivač na strukturu tipa FILE, tj. tok podataka ("struju") `stream`, (npr. stdin, stdout ili neki drugi pokazivač tipa FILE *). Osim što je pridruženi pokazivač `stream` – argument funkcije, on je i njena povratna vrednost, tj. funkcija vraća tok podataka `stream`, ili NULL ako dođe do greške. Funkcija `freopen` se obično koristi da promeni datoteku koja je pridružena pokazivačima standardnog ulaza i izlaza – stdin i stdout, odnosno kao alternativa operatora preusmerenja.

Primer 1

```
#include <stdio.h>
int main()
{
    int i;
    FILE *ulazp, *izlazp;
    ulazp=freopen("ulaz.txt", "r", stdin);
    izlazp=freopen("izlaz.txt", "w", stdout);
    scanf("%d", &i);
```

```
    printf("%d \n", i);
    return 0;
}
```

Program čita celobrojnu vrednost promenljive *i* iz datoteke ulaz.txt i upisuje je u datoteku izlaz.txt, pridružujući te datoteke standardnom ulazu odnosno standardnom izlazu funkcijom freopen(). Pošto je ovo pridruživanje realizovano pozivima funkcije freopen, njena povratna vrednost se i ne koristi, pa je efekat progaram isti kao i u sledećem primeru:

Primer 2

```
#include <stdio.h>
int main()
{
    int i;
    freopen("ulaz.txt", "r", stdin);
    freopen("izlaz.txt", "w", stdout);
    scanf("%d", &i);
    printf("%d \n", i);
    return 0;
}
```

2.4 Argumenti u komandnoj liniji

Primer iz tačke 2.2 odnosio se na kopiranje datoteke a.c u datoteku b.c. Da bismo poopštili rešenje (na kopiranje datoteke x u datoteku y), možemo da iskoristimo svojstvo C-a da se u komandnoj liniji kojom se izvršava .exe program navedu argumenti programa, u našem slučaju – ime datoteke koja se kopira i ime datoteke u koju se kopira.

Funkcija main do sada je definisana bez argumenata – main(), a komandna linija kojom se poziva izvršavanje programa je sadržala samo jedan "argument" - naziv samog izvršnog programa.

U opštem slučaju, funkcija main može da ima argumente, a njihove vrednosti u tesnoj su vezi sa brojem i vrednostima argumenata komandne linije kojom se poziva izvršavanje odgovarajućeg programa. Dakle, u opštem slučaju funkcija main se definiše sa dva argumenta: argc (ARGument Count) – broj argumenata u komandnoj liniji kojom se poziva izvršavanje programa, i argv (ARGument Vector) – niz pokazivača na stringove (niske karaktera) koji sadrže argumente komandne linije, po jedan za svaki string. Ako se program poziva samo imenom (u komandnoj liniji nema dodatnih argumenata), onda je broj argumenata komandne linije, tj. vrednost argumenta argc funkcije main jednak 1, dok je argv[0] pokazivač na ime exe-datoteke, a argv[1] je NULL.

Ako se program poziva navođenjem dodatnih argumenata u komandnoj liniji (pored samog imena izvršnog programa), funkcija main definiše se sa argumentima argc, argv, (main(int argc, char *argv[ ])), pri čemu argc dobija vrednost

jednaku broju argumenata (računajući i sámo ime datoteke izvršnog programa – to je argv[0]), argv[1] jednak je prvom opcionom argumentu (koji se navodi iza imena izvršne datoteke), argv[2] jednak je drugom opcionom argumentu, itd.

Sada program iz tačke 2.2 može da ima dva opciona argumenta – ime ulazne i ime izlazne datoteke, i sledeći oblik:

```
#include <stdio.h>
/*kopiranje: kopira ulaznu datoteku u izlaznu datoteku */
main(int argc, char *argv[ ])
{
    FILE *ulazp, *izlazp;
    void filecopy(FILE *, FILE *);
    if ((ulazp=fopen(*++argv, "r"))==NULL) {
        printf("ne moze da se otvori datoteka %s \n", *argv);
        return 1;
    }
    else {
        if ((izlazp=fopen(*++argv, "w"))==NULL) {
            printf("ne moze da se otvori datoteka %s\n", *argv);
            return 1;
        }
        else {
            filecopy(ulazp, izlazp);
            fclose(izlazp);
            izlazp=fopen(*argv, "r");
            filecopy(izlazp, stdout);
            fclose(izlazp);
        }
        fclose(ulazp);
    }
}
```

Ako se program nalazi u datoteci datkopi, onda se njegovo izvršavanje kojim se datoteka a.c kopira u datoteku b.c poziva komandnom linijom oblika

datkopi a.c b.c

2.5 Funkcije direktnog pristupa datotekama

Do sada je podrazumevan sekvencijalni pristup datotekama, odnosno, da bi se pročitao n-ti red tekstualne datoteke, trebalo je pročitati (pristupiti) i prethodnih n-1 redova. Takođe, moguće je bilo upisati novu vrednost samo na kraj datoteke otvorene za upis ili dodavanje. Ako je datoteka smeštena na medijumu sa direktnim pristupom (npr. na disku) C omogućuje i direktan pristup pojedinim bajtovima sa mogućnošću čitanja ili upisa, u zavisnosti od načna na koji je datoteka otvorena.

Pomenućemo dve funkcije direktnog pristupa, `fseek()` i `ftell()`, obe definisane u standardnom zaglavlju <stdio.h>.

Funkcija fseek ima zaglavlje oblika

```
int fseek(FILE *stream, long offset, int origin)
```

i postavlja indikator pozicije pridružen toku (pokazivaču) `stream` na novu poziciju koja se određuje dodavanjem `offset` bajtova na referentnu poziciju određenu `origin`-om.

Vrednost `origin` može biti SEEK_SET (početak datoteke), SEEK_CUR (tekuća pozicija u datoteci) ili SEEK_END (kraj datoteke). Ove vrednosti su konstante definisane u <stdio.h>.

Za tekstualnu dastoteku, `offset` treba da bude 0 ili vrednost koju vraća funkcija ftell, kada `origin` treba da bude SEEK_SET, mada neki kompilatori ne primenjuju ovo ograničenje.

Funkcija fseek vraća vrednost 0 ako je pozicioniranje uspešno, a vrednost različitu od 0 u slučaju greške.

Primer

```
#include <stdio.h>
int main()
{
    FILE *fp;
    fp=fopen("izlaz.txt", "w");
    fputs("This is an apple.", fp);
    fseek(fp, 9, SEEK_SET);
    fputs(" sam", fp);
    fclose(fp);
    return 0;
}
```

Program upisuje u datoteku izlaz.txt prvo tekst "This is an apple." a zatim od pozicije 10 (pozicije koja je 9 bajtova udaljena od početka datoteke tj. od prvog karaktera) upisuje tekst " sam" pa je rezultat u datoteci izlaz.txt tekst "This is a sample.".

Primer

```
#include <stdio.h>
int main()
{
    FILE *fp;
    fp=fopen("ulaz.txt", "r");
    fseek(fp, 9, SEEK_SET);
    printf("%c\n", getc(fp));
    fclose(fp);
    return 0;
}
```

Program ispisuje na standardni izlaz 10. karakter iz datoteke ulaz.txt.

Funkcija ftell ima zaglavlje oblika

```
long int ftell(FILE *stream)
```

i vraća tekuću vrednost indikatora pozicije u datoteci kojoj je pridružen pokazivač `stream`.

Za binarnu datoteku, vraćena vrednost odgovara broju bajtova od početka datoteke. Za tekstualnu datoteku, ne može da se garantuje da će vrednost biti tačan broj bajtova od početka datoteke, ali vraćena vrednost može da se koristi za pozicioniranje u drugoj funkciji, fseek.

Argument `stream` (struja) je pokazivač na strukturu tipa FILE koja identifikuje datoteku, dok je povratna vrednost, u slučaju uspeha, tekuća vrednost indikatora pozicije, a u slučaju greške -1L.

Primer

```
#include <stdio.h>
int main()
{
    FILE *fp;
    long size;
    fp=fopen("ulaz.txt", "r");
    if(fp==NULL){
        printf("greska pri otvaranju datoteke");
        return 1;
    }
    else
    {
        fseek(fp, 0, SEEK_END);
        size = ftell(fp);
        fclose(fp);
        printf("Velicina datoteke ulaz.txt: %ld bajtova \n", size);
        return 0;
    }
}
```

Program određuje veličinu datoteke u bajtovima tako što se funkcijom fseek pozicionira na kraj datoteke a onda funkcijom ftell određuje koliko je ta pozicija udaljena od početka datoteke.

3

Operatori nad bitovima

Pored uobičajenih (prethodno pomenutih) aritmetičkih, relacijskih i logičkih operatora, programski jezik C ima i operatore nad bitovima (primenjuju se samo na celobrojne argumente).

Operatori za manipulisanje pojedinačnim bitovima u C-u su:

& – AND nad bitovima (u rezultatu postavlja 1 samo u one bitove u kojima oba operanda imaju 1);

| – inkluzivni OR nad bitovima (postavlja 1 u sve one bitove u kojima bar jedan od operanada ima 1);

^ – ekskluzivni OR nad bitovima (postavlja 1 samo u one bitove u kojima tačno jedan operand ima 1);

<< – levi šift (pomeranje levog operanda ulevo za broj bitova sadržan u desnom operandu; u oslobođene bitove upisuje se 0; analogno sa množenjem odgovarajućim stepenom dvojke)

>> – desni šift (pomeranje levog operanda udesno za broj bitova sadržan u desnom operandu; najviši bit se ili ponavlja (aritmetički šift) ili se u njega upisuje 0 – logički šift).

Ukoliko je desni operand šift operatora negativan (ili veći od broja bitova levog operanda), dejstvo operatora je nedefinisano.

~ – jednostruki komplement (bitovi 1 zamenjuju se sa 0, a 0 sa 1).

Operator AND nad bitovima koristi se za maskiranje nekih bitova. Na primer,

```
n = n & 0177;
```

postavlja na 0 sve bitove osim najnižih 7 bitova od n.

Dodela

```
x = x & ~077;
```

izvršava se tako što se prvo operatorom jednostrukog komplementa u konstanti 077 postavljaju 0-bitovi na 1 a 1-bitovi na 0, a zatim se u x ostavljaju samo 1-bitovi iz gornjih 10 pozicija (donjih 6 su bili 1-bitovi i postali su 0-bitovi).

Dosad opisani oparatori mogu se grupisati po prioritetima u sledeće grupe i poređati na sledeći način (prioritet opada sa porastom rednog broja):

1.	(), []	sleva nadesno
2.	!, ~, ++,−−, +, −, ∗, &, (*tip*)	**zdesna nalevo**
3.	∗, /, %	sleva nadesno
4.	+, −	sleva nadesno
5.	**<<, >>**	**sleva nadesno**
6.	<, <=, >, >=	sleva nadesno
7.	==, ! =	sleva nadesno
8.	**&**	**sleva nadesno**
9.	^	**sleva nadesno**
10.	|	**sleva nadesno**
11.	&&	sleva nadesno
12.	||	sleva nadesno
13.	?:	zdesna ulevo
14.	=, +=, −=, /=, %=, &=, ^=, |=, <<=, >>=	zdesna ulevo
15.	,	sleva udesno

Primer Napisati program koji razmenjuje zadati par bitova u učitanom broju. Bitovi koje treba razmeniti zadaju se svojim rednim brojem kao argumenti komandne linije.

```
#include <stdio.h>
unsigned Trampa(unsigned n, int i, int j);
main(int argc, char **argv)
{
    unsigned x; /*broj sa standardnog ulaza ciji se bitovi razmenjuju*/
    int i,j; /*pozicije bitova za trampu*/
    /*ocitavanje argumenata komandne linije i broja sa standarnog ulaza*/
    sscanf(argv[1], "%d", &i);
    sscanf(argv[2], "%d", &j);
    scanf("%u", &x);
    printf("\n Nakon trampe vrednost unetog broja je %u\n", Trampa(x,i,j));
}

unsigned Trampa(unsigned n, int i, int j)
{
//ako se bit na poziciji i razlikuje od bita na poziciji j, treba ih invertovati
    if ( ((n>>i)&1) != ((n>>j)&1) ) n^= (1<<i) | (1<<j);
    return n;
}
```

4

Analiza algoritama - vremenska i prostorna složenost

Klasa zadataka istog tipa definiše se kao problem. Za većinu problema postoji mnogo različitih algoritama i potrebno je izabrati najbolju implementaciju od svih tih algoritama. Najbolja implementacija podrazumeva minimalnu potrošnju resursa kao što su memorija i vreme.

Problemi koje rešavamo imaju prirodnu "veličinu" ("dimenziju"), koja obično predstavlja količinu podataka koje treba obraditi. Složenost algoritma (vremsnska ili prostorna) je obično neka funkcija koja povezuje veličinu problema (ulaza) sa brojem koraka izvršavanja algoritma (vremenska složenost) ili brojem potrebnih memorijskih lokacija (prostorna složenost). U algoritmima sortiranja ili pretraživanja veličina ulaza biće broj elemenata koje treba sortirati ili koje treba pretražiti, N. U algoritmima nad celim brojevima to može da bude apsolutna vrednost tih brojeva - na primer, kod Euklidovog algoritma za nalaženje najvećeg zajedničkog delioca celih brojeva a, b, dimenzija problema može da bude $\max(|a|, |b|)$. Dakle, potrebno je odrediti (proceniti) resurse (vreme i/ili memoriju) koji se troše za rešavanje problema, kao funkciju od N, odnosno od a (b), i sl. Zainteresovani smo, kako za prosečni slučaj (količinu vremena koje se utroši pri "tipičnim" ulaznim podacima), tako i za najgori slučaj (vreme potrebno za rešavanje problema pri najgoroj mogućoj ulaznoj konfiguraciji).

Uobičajeno je da se složenost algoritama (vremenska ili prostorna) procenjuje u asimptotskom smislu, tj. da se funkcija složenosti procenjuje za dosta velike dužine ulaza. Za to se koriste "veliko O" notacija ($\mathrm{O}()$), "omega notacija" ($\Omega()$), i "teta notacija" ($\Theta()$).

"Veliko O" notacija, poznata kao Landau ili Bahman-Landau notacija, opisuje granično ponašanje funkcije kada argument teži nekoj specifičnoj vrednosti ili

beskonačnosti, obično u terminima jednostavnijih funkcija. Formalno, pretpostavimo da su $f(x)$ i $g(x)$ dve funkcije definisane na nekom podskupu skupa realnih brojeva. Kažemo da je

$f(x) = O(g(x))$ kad $x \to \infty$

(koristi se i notacija $f(x) \in O(g(x))$)
ako i samo ako postoji pozitivan realni broj c i realni broj x_0 takav da je

$|f(x)| \leq c \cdot |g(x)|$ za $x > x_0$.

Na primer, ako je $n \in N$ i ako je $f(n) = 2n^2 + n - 2$, onda je $f(n) = O(n^2)$ ali i $f(n) = O(n^3)$ (odrediti c i n_0 iz definicije $O(...)$).

Slično O-notaciji definišu se i Ω i Θ asimptotske notacije.

$f(x) \in \Omega(g(x))$ ako i samo ako $\exists(c > 0), x_0 : \forall(x > x_0)\ |f(x)| \geq c \cdot |g(x)|$

$f(x) \in \Theta(g(x))$ ako i samo ako $\exists(c_1, c_2 > 0), x_0 : \forall(x > x_0)\ c_1 \cdot |g(x)| < |f(x)| < c_2 \cdot |g(x)|$ (slika 4.1).

"Veliko O" notacija - dobila je ime od "order of" ili "red veličine" pa ćemo za vremensku složenost koja je reda $O(f(n))$ govoriti da je "proporcionalna" sa $f(n)$.

Analiza vremenske složenosti algoritma ne mora biti precizna (da prebroji svaki korak u izvršavanju algoritma), već je dovoljno da odredi najveće elemente takvih proračuna. Na primer, ako bismo strogim matematičkim aparatom došli do vremena $((12 \ln 2)/\pi^2) \ln a$ za izvršavanje rekurzivnog Euklidovog algoritma ($a > b$), za kompleksnost tog algoritma dovoljno je (po mogućstvu jednostavnijom metodom) proceniti da je njegovo vreme izvršavanja proporcionalno sa $\ln a$.

Većina algoritama sa kojima se srećemo ima vreme izvršavanja proporcionalno jednoj od sledećih funkcija ($O(\dots)$):

1: Ako se svi iskazi programa izvršavaju samo po jednom ili najviše po nekoliko puta (nezavisno od veličine i broja ulaznih podataka), kaže se da je vreme izvršavanja tog programa konstantno. To je očigledno najbolja moguća vrsta algoritama.

Ovakvu vremensku složenost ima, na primer, algoritam direktnog pretraživanja u idealnom slučaju:

Pretpostavljamo da je dat niz elemenata $a_1, a_2, \dots, a_N$, koji se upisuju u tablicu dužine N i to tako da se element a_i upisuje u vrstu tablice j koja se sračunava nekom (dobro odabranom) funkcijom h direktnog pristupa: $j = h(a_i)$, $1 \leq j \leq N$. Ako se, za zadatu vrednost v, pretražuje tablica da bi se utvrdilo da li je u njoj prisutan element niza koji je jednak v, onda se izračuna vrednost $j = h(v)$ i vrednost v uporedi sa sadržajem j-te vrste tablice. Ako su vrednosti jednake, pretraživanje je uspešno, inače vrednost v ne postoji u tablici. Vreme pretraživanja jednako je vremenu izračunavanja funkcije h za argument v, i ono je konstantno – ne zavisi od broja elemenata niza.

log$N$: Ako vreme izvršavanja programa lagano raste (brzina se lagano usporava) kako N raste, kaže se da je njegovo vreme izvršavanja logaritamsko. Za

naša razmatranja može se smatrati da je vreme izvršavanja manje od neke "velike" konstante. Ako je osnova logaritma 10, za $N = 1000$, $\log N$ je 3, za $N = 1.000.000$, $\log N$ je samo dva puta veće, tj. 6. (Ako je osnova logaritma 2, za $N = 1000$, $\log N$ je približno 10, što je veća konstanta od 3, ali nebitno veća; osnova logaritma je kod ove klase algoritama inače nebitna jer se svi oni ponašaju isto do na konstantu: $\log_a N = \log_b N * \log_a b$). Bez obzira na osnovu, kadgod se N udvostruči, $\log N$ poraste za konstantu, a $\log N$ se udvostruči tek kad se N kvadrira.

Ovakvu vremensku složenost ima, na primer, algoritam binarnog pretraživanja.

N: Kada se mala količina obrade izvrši nad svakim ulaznim podatkom, vreme izvršavanja programa je linearno. Kada je N milion, vreme izvršavanja je neka konstanta puta milion. Kada se N udvostruči, udvostruči se i vreme izvršavanja. To je optimalno vreme izvršavanja za algoritam koji mora da obradi N ulaznih podataka ili da proizvede N izlaznih rezultata (npr. da učita vrednosti N elemenata nekog niza i da ih odštampa).

$N\log N$: Vreme proporcionalno sa $N\log N$ utroše za svoje izvršavanje algoritmi koji problem rešavaju tako što ga razbiju u manje potprobleme, reše te potprobleme nezavisno jedan od drugog i zatim kombinuju rešenja. Kada je N milion, $N\log N$ je možda dvadeset miliona. Kada se N udvostruči, vreme izvršavanja se poveća za više nego dvostruko (ali ne mnogo više).

Ovakvu vremensku složenost ima, na primer, algoritam brzog sortiranja niza (quicksort) (v. odeljak 7.1).

N^2: Kvadratno vreme izvršavanja nastaje za algoritme koji obrađuju sve parove ulaznih podataka (najčešće u dvostruko ugnježdenoj petlji). Algoritam sa ovakvim vremenom izvršavanja praktično je primenjivati samo za relativno male probleme (malo N). Kada je N hiljadu, vreme izvršavanja je milion. Kada se N udvostruči, vreme izvršavanja se uveća četiri puta.

Ovakvu vremensku složenost imaju elementarni metodi sortiranja, na primer sortiranje izborom.

N^3: Slično, algoritam koji obrađuje trojke podataka (npr. u trostruko ugnježdenoj petlji) ima kubno vreme izvršavanja i praktičan je za upotrebu samo za male probleme. Kada je N sto, vreme izvršavanja je milion. Kadgod se N udvostruči, vreme izvršavanja se uveća osam puta.

2^N: Algoritmi sa eksponencijalnim vremenom izvršavanja nastaju prirodno pri primeni rešenja "grubom silom", tj. rešenja koja obično prvo padnu na pamet. Ipak, za neke probleme nisu nađeni (a sumnja se i da će ikada biti nađeni) algoritmi manje vremenske složenosti. Ovakvi algoritmi su vrlo retko praktični za upotrebu. Kada je $N = 20$, vreme izvršavanja je milion. Kada se N udvostruči, vreme izvršavanja se kvadrira.

Vreme izvršavanja nekog programa tj. algoritma je obično neka konstanta puta jedan od navedenih izraza (glavni izraz), plus neki manji izraz. Pri velikom N,

efekat glavnog izraza dominira. Za mala N, i manji izrazi mogu značajno da doprinesu ukupnom vremenu izvršavanja, pa je analiza složenija. Zato se pri formulama složenosti koje se izvode podrazumeva veliko N.

Slično vremenskoj složenosti, prostorna složenost algoritma odnosi se na funkciju zahtevanog memorijskog prostora za izvršenje tog algoritma, u zavisnosti od veličine ulaza. Zapravo, prostorna složenost i odgovarajuća funkcija obično se odnose na dodatni memorijski prostor (pored prostora neophodnog za smeštanje samih ulaznih podataka) potreban za izvršenje algoritma. Meri se takođe velikom O notacijom (O()), "omega notacijom" ($\Omega()$), i "teta notacijom" ($\Theta()$) i može da pripada prethodno opisanim klasama složenosti.

Na primer, alogoritam sortiranja niza umetanjem (ili izborom) koristi samo prostor za elemente tog niza (i još nekoliko memorijskih lokacija čiji broj ne zavisi od broja elemenata niza), pa je njegova (dodatna) prostorna složenost O(1).

S druge strane, algoritam brzog sortiranja (quicksort) niza od N brojeva, pored N lokacija za smeštanje tih brojeva, zahtevaće još $\log N$ memorijskih lokacija (za obradu rekurzije, v. odeljak 7.1), pa je njegova (dodatna) prostorna složenost O($\log N$).

Slično, algoritam sortiranja spajanjem (merge sort), za niz od N elemenata zahteva dodatnih N lokacija za premeštanje elemenata i još $\log N$ lokacija za obradu rekurzije, pa je njegova (dodatna) prostorna složenost O(N)+O($\log N$) = O(N) (v. odeljak 7.3; postoji i implementacija sa dodatnom prostornom složenošću O($\log N$)).

U opštem slučaju, postoji nagodba između prostora i vremena koje jedan problem zahteva za svoje rešavanje: on se ne može rešiti i za kratko vreme i uz malo korišćenje prostora. Neki algoritmi za rešavanje tog problema koristiće malo vremena i puno prostora, drugi obratno. Obično se pravi neki kompromis u zavisnosti od izabranog algoritma.

Najčešća greška pri izboru algoritma je zanemarivanje karakteristika efikasnosti. Često je moguće naći neznatno komplikovaniji algoritam koji radi u vremenu $N\log N$ (mnogo brži) od jednostavnog algoritma koji radi u vremenu N^2 (mnogo sporiji) - i slično u odnosu na prostor. Takođe je greška i pridati preveliki značaj efikasnosti, jer bolji algoritam može da bude znatno kompleksniji a da postiže bolje rezultate samo za neke vrlo velike N, ili da se izvršava ograničen broj puta pri čemu je vreme potrebno za realizaciju kompleksnijeg algoritma veće od vremena izvršavanja jednostavnijeg.

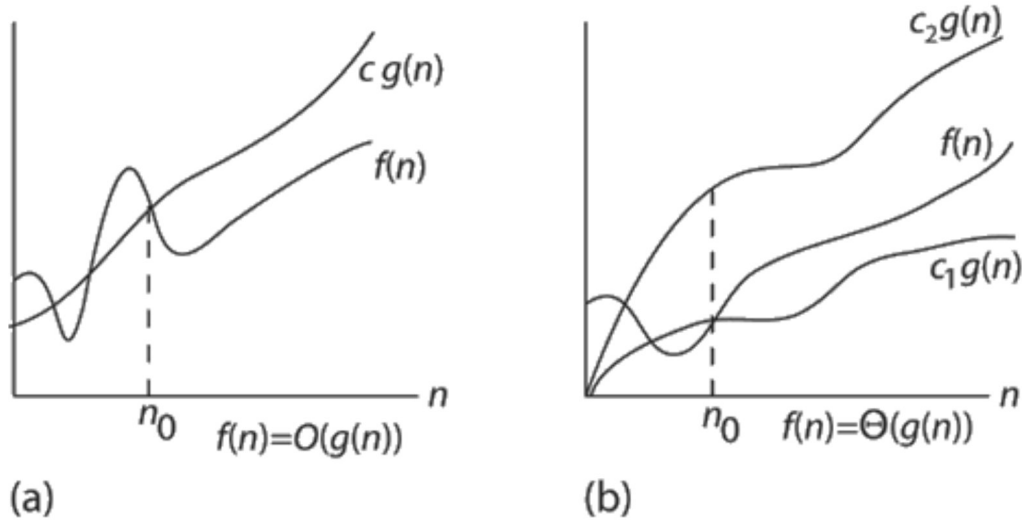


Figure 4.1: O i Θ notacija: (a) $f(n) = O(g(n))$ (b) $f(n) = \Theta(g(n))$

5

Elementarne metode sortiranja

Elementarne metode sortiranja su jednostavne ali za velike skupove neefikasne. Ta neefikasnost je posledica činjenice da je njihova vremenska složenost proporcionalna sa n^2 gde je n – kardinalnost skupa.

Elementarne metode sortiranja stoga daju dobre rezultate za male skupove slogova (brojeva) (do 500). Kada se sortiranje izvodi jedanput ili ograničen broj puta, i elementarni algoritmi mogu biti zadovoljavajući.

Elementarne metode sortiranja, između ostalog, omogućuju izučavanje složenijih. Takve su, na primer, metode sortiranje izborom najmanjeg (najvećeg) elementa, sortiranje umetanjem, šelsort (modifikacija sortiranja umetanjem), bubble sort, itd.

5.1 Sortiranje izborom najvećeg elementa

Sortiranje niza u neopadajućem ili nerastućem poretku podrazumeva nalaženje jedne permutacije elemenata niza u kojoj se elementi pojavljuju u neopadajućem tj. nerastućem poretku.

Metoda sortiranja izborom najvećeg elementa odnosi se na sortiranje niza podataka x sa n elemenata u nerastući poredak (slično izbor najmanjeg elementa obezbeđuje sortiranje u neopadajući poredak). Prvo se nalazi najveći element niza i on se "dovodi" na prvo mesto, zatim se nalazi najveći od preostalih $n-1$ elemenata i on se "dovodi" na drugo mesto, nalazi najveći od preostalih $n-2$ elemenata i dovodi na treće mesto, itd, zaključno sa nalaženjem većeg od poslednja dva elementa i njegovim "dovođenjem" na pretposlednje mesto. Na poslednjem mestu će ostati element koji nije veći ni od jednog u nizu (najmanji element). "Dovođenje" najvećeg elementa na prvo mesto najjednostavnije se realizuje razmenom mesta tog elementa sa prvim elementom. Da bi se znalo sa kog mesta "dolazi" najveći element (i na koje treba "poslati" prvi element), potrebno je pamtiti indeks najvećeg elementa. Slično važi i za "dovođenje" najvećeg elementa preostalog niza na drugo,

treće, itd, mesto.

Dakle, globalna struktura algoritma ima sledeći oblik:

(a) odrediti najveći element x_{max} prema prethodnom algoritmu nalaženja maksimalnog elementa;
(b) razmeniti x_{max} i x_1 i
(c) ponoviti korake (a) i (b) posmatrajući nizove $x_2, \ldots, x_n; x_3, \ldots, x_n; \ldots$, dok ne ostane samo x_n, tj.

```
for (i=1; i<n; i++)
{
    naći najveći element u nizu x[i], x[i + 1], ..., x[n] i zapamtiti njegov indeks
max;
    razmeniti x[i] i x[max];
}
```

Nalaženje maksimalnog elementa niza. Neka je dat niz $x[m..n]$. Zadatak je naći indeks j t.d. $x_j = \max(x_m, \ldots, x_n)$.

Nalaženje najvećeg elementa u nizu čiji su elementi na "mestima" (indeksima) $m, m + 1, \ldots, n$ može se realizovati tako što se prvo prvi u nizu proglasi za najveći (max $= m$), a zatim se najveći poredi sa svim sledećim elementima (sa indeksima od $m + 1$ do n). Kadgod se naiđe na element veći od najvećeg, on se proglašava za najveći tj. njegov indeks, j, dodeljuje se indeksu najvećeg elementa, max (max $= j$).

Funkcija za nalaženje najvećeg elementa u nizu $x[m..n]$ celih brojeva, u C-u ima sledeći oblik:

```
int maxelem(int x[ ], int m, int n)
{
    int max;
    max = m;
    for (j=m+1; j <= n; j++)
        if (x[j] > x[max]) max=j;
    return max;
}
```

Funkcija za sortiranje izborom najvećeg elementa. Kako smo razvili i funkciju za nalaženje najvećeg elementa i funkciju razmene (swap), možemo napisati i funkciju za sortiranje niza (celih brojeva) izborom najvećeg elementa u C-u (uz odgovarajuće pomeranje indeksa s obzirom da indeksi kreću od 0):

```
void maxsort(int x[ ], int n)
{
    void swap(int x[ ], int i, int j);
    void maxelem(int x[ ], int m, int n);
    int i,j,max;
    for(i=0; i<n–1; i++)
```

```
            swap(x, i, maxelem(x, i, n–1));
}
void swap(int x[ ], int i, int j)
{
    int priv;
    priv = x[i];
    x[i] = x[j];
    x[j] = priv;
}
```

Ovaj algoritam pokazuje dobra svojstva za male nizove, npr. do 1000 elemenata. Broj poređenja između elemenata niza je oko $n^2/2$ jer se spoljašnja petlja (po i) izvršava n puta a unutrašnja (po j) oko $n/2$ puta u proseku. Iskaz max= j se izvršava reda veličine $n\log n$ puta pa nije deo unutrašnje petlje. Stoga je vreme izvršavanja ovog algoritma proporcionalno sa n^2.

5.2 Sortiranje umetanjem

Ako je dat niz (x_n) sa elementima nekog, uređenog tipa T, koji treba urediti u neopadajući poredak, ova metoda sortiranja polazi od pretpostavke da imamo uređen početni deo niza, $x_1, \ldots, x_{i-1}$ (to svakako važi za $i = 2$, jer je podniz sa jednim elementom uređen); u svakom koraku, počevši od $i = 2$ i povećanjem i, i-ti element se stavlja na pravo mesto u odnosu na prvih (uređenih) $i - 1$.

Na primer, slika 5.1 prikazuje postupak sortiranja datog niza od 8 celobrojnih elemenata:

Algoritam sortiranja umetanjem prolazi kroz sledeće korake:
for (i=2; i<=n; i++) {
    $v = x[i]$;
    "umetnuti v na odgovarajuće mesto u $x_1, x_2, \ldots, x_i$
}
"Umetanje na odgovarajuće mesto" može se izvršiti pomeranjem elemenata koji prethode $(x[j], j = i-1, \ldots, 1)$ za po jedno mesto udesno $(x[j+1] = x[j])$. Kriterijum kraja tog pomeranja je dvojak:

1. nađena je vrednost x_j t.d. $x_j < v$ (onda se v stavlja na $j + 1$. mesto);
2. došlo se do levog kraja niza

Funkcija na C-u koja realizuje ovu metodu ima sledeći oblik (uz odgovarajuće pomeranje indeksa s obzirom da indeksi kreću od 0):

```
void umetsort(int x[ ], int n)
{
    int v;
    for(i=1; i<n; i++) {
        v=x[i]; j=i–1;
```

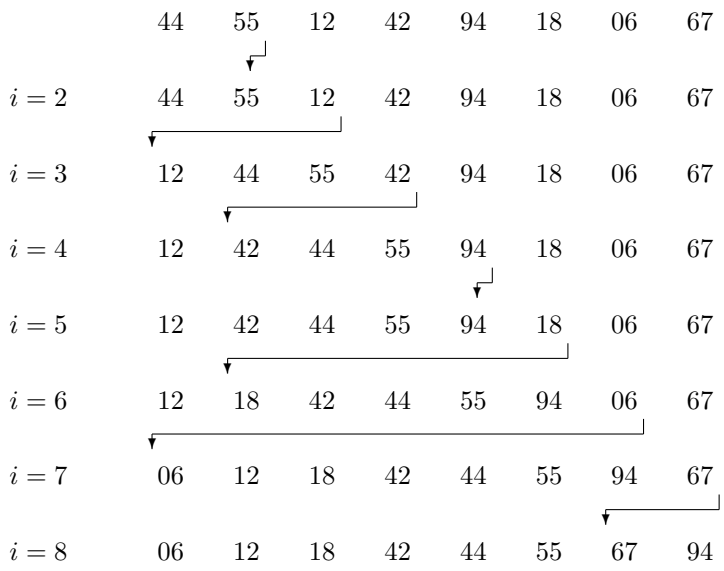


Figure 5.1: Primer sortiranja umetanjem

```
            while (j>=0 && v<x[j]) {
                x[j+1]=x[j]; j--;
            }
            x[j+1]=v;
        }
    }
```

Broj poređenja, P_i, u i-tom prolazu je najviše i a najmanje 1, a u proseku je $i/2$. Broj premeštanja, M_i, je $P_i + 2$. Zato je ukupan broj poređenja i premeštanja

$P_{min} = n - 1$

$P_{max} = n + (n-1) + (n-2) + \cdots + 1 = n \cdot (n+1)/2 = \frac{1}{2}(n^2 + n)$,

i slično za M_i.

Najmanji broj poređenja i premeštanja je kada je polazni niz uređen; najveći je kada je polazni niz uređen u obrnutom poretku.

Moguća su poboljšanja algoritma koja koriste uređenost niza $x_1, x_2, \ldots x_{i-1}$ (npr. binarno pretraživanje kojim se dobije **sortiranje binarnim umetanjem**).

Za vežbu napisati odgovarajuću funkciju sortiranja binarnim umetanjem.

5.3 Šelsort

Šelsort je jednostavno proširenje sortiranja umetanjem koje dopušta direktnu razmenu udaljenih elemenata. Proširenje se sastoji u tome da se kroz algoritam

umetanja prolazi više puta; u prvom prolazu, umesto koraka 1 uzima se neki korak h koji je manji od n (što omogućuje razmenu udaljenih elemenata) i tako se dobija h-sortiran niz, tj. niz u kome su elementi na rastojanju h sortirani, mada susedni elementi to ne moraju biti. U drugom prolazu kroz isti algoritam sprovodi se isti postupak ali za manji korak h. Sa prolazima se nastavlja sve do koraka $h = 1$, u kome se dobija potpuno sortirani niz.

Dakle, ako se svako pojavljivanje "1" u algoritmu sortiranja umetanjem zameni sa h, dobija se algoritam koji, uzimajući svaki h-ti element (počevši od bilo kog) proizvodi sortirani niz. Takav, "h-sortirani" niz je, u stvari, h sortiranih podnizova, koji se preklapaju. h-sortiranjem za veliko h premeštaju se udaljeni elementi, i time se olakšava h-sortiranje za manje vrednosti h. Koristeći ovu proceduru za bilo koji niz vrednosti od h, koji se završava sa 1, proizvodi se sortirani niz. To je šelsort.

Za najveći (prvi) korak h zgodno je uzeti (mada ne i optimalno u odnosu na vreme izvršavanja algoritma) najveći broj oblika $3 \times k + 1$ koji nije veći od n, kao sledeći korak h – celobrojnu trećinu prethodnog h, itd, sve dok se za h ne dobije vrednost 1. Moguće je za h uzeti i druge sekvence, na primer $n/2$, $n/2^2$, $n/2^3$, ..., 1.

Na primer, za petnaest elemenata u nizu (n=15), za h=13,4,1, redom, dobiju se sledeći nizovi:

vr. indeksa:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
početni niz:	P	R	I	M	E	R	S	O	R	T	I	R	A	NJ	A
$h = 13$:	NJ	A	I	M	E	R	S	O	R	T	I	R	A	P	R
$h = 4$:	A	A	I	M	E	P	I	O	NJ	R	R	R	R	T	S
$h = 1$:	A	A	E	I	I	M	NJ	O	P	R	R	R	R	S	T

U prvom prolazu, P sa pozicije 1 se poredi sa NJ sa pozicije 14, i razmenjuju se, zatim se R sa pozicije 2 poredi sa A sa pozicije 15 i razmenjuju se. U drugom prolazu, NJ, E, R, A sa pozicija 1,5,9,13, redom, razmeštaju se tako da daju redosled A, E, NJ, R na tim pozicijama, i slično za pozicije 2, 6, 10, 14, itd. Poslednji prolaz je baš sortiranje umetanjem, ali se nijedan element ne premešta daleko. Izbor za h je proizvoljan i ne postoji dokaz da su neki izbori za h optimalni. Niz koji se u praksi pokazao dobar je ...,1093, 364, 121, 40, 13, 4, 1. Funkcija za šelsort, kao i prethodne funkcije sortiranja, ima zaglavlje oblika:

void shellsort(int x[ ], int n)

Za kompletnu definiciju funkcije shellsort v. Kernighan, Ritchie, "The C programming language" – tačka 3.5.

Metodu shellsort je teško porediti sa drugim zato što je funkcionalni oblik vremena izvršavanja nepoznat i zavisi od sekvence za h. Za gornji algoritam su granice $n*(\log n)^2$ i $n^{1.25}$. Metoda nije posebno osetljiva na početni redosled elemenata, za razliku od algoritma sortiranja umetanjem koji ima linearno vreme izvršavanja ako je niz u početku sortiran, i kvadratno vreme ako je niz u obrnutom redosledu. Metoda se često primenjuje jer daje zadovoljavajuće vreme izvršavanja do, npr. 5000 elemenata, a algoritam nije složen.

5.4 Bubble sort

Ova metoda je elementarna: ponavlja se prolazak kroz niz elemenata i razmenjuju se susedni elementi, ako je potrebno, sve dok se ne završi prolaz koji ne zahteva nijednu razmenu. Tada je niz sortiran.

```
void bubblesort(int x[ ], int n)
{
    void swap(int x[ ], int i, int j);
    int j, ind;
    do {
        ind=0;
        for(j=1; j<n; j++)
            if (x[j–1] > x[j]) {
                swap(x, j–1, j);
                ind = 1;
            }
    } while (ind);
}
```

Algoritam bubble sort ima kvadratno vreme, složeniji je i manje efikasan od sortiranja umetanjem. Unutrašnja petlja ima više iskaza i od sortiranja umetanjem i od sortiranja izborom.

6

Rekurzija

Neformalno govoreći, ukoliko se u definiciji funkcije poziva ta ista funkcija (direktno ili indirektno), kaže se da je funkcija **rekurzivna**. Struktura definicije rekurzivne funkcije uključuje dve komponente: uslov pri kome se funkcija rekurzivno poziva (i sámo to pozivanje), i uslov završetka (i završetak) izvršavanja funkcije – izlaz iz rekurzije.

Potreba za rekurzivnom kompozicijom radnji je česta i uglavnom vezana za probleme čija je formulacija rekurzivna.

Na primer, formula za računanje faktorijela prirodnog broja (ili 0) može biti iterativna:

$$n! = \begin{cases} 1 & \text{ako je } n = 0, \\ 1 * 2 * \cdots * (n-1) * n & \text{ako je } n > 0 \end{cases}$$

a može biti zadata i rekurzivno:

$$n! = \begin{cases} 1, \text{ako je } n = 0, \\ n * (n-1)!, \text{ako je } n > 0 \end{cases}$$

Dok deklaracija iterativne funkcije za izračunavanje faktorijela ima oblik

```
int fakti(int n)
{
    int i;
    int r;
    r=1;
    for(i=1; i<=n; i++) r=r*i;
    return r;
}
```

deklaracija rekurzivne funkcije ima sledeći oblik:

```
int faktr(int n)
```

```
{
    int w;
    if(n==0) w=1;
    else w=n*faktr(n-1);
    return w;
}
```

Pri svakom pozivu funkcije, kreira se primerak svake unutrašnje (lokalne) promenljive. Taj primerak prestaje da postoji, tj. oslobađa se memorijski prostor koji je zauzimao, posle završetka izvršavanja tog poziva funkcije (posle završetka tog "aktiviranja" funkcije).

Na primer, u toku izvršavanja rekurzivne funkcije faktr sa argumentom 5 – po pozivu faktr(5), kreira se šest lokalnih promenljivih w od kojih svaka, u pozivu u kome je kreirana, dobija vrednost faktorijela broja za koji je funkcija pozvana, zatim se ta vrednost vraća prethodnom pozivu funkcije, i umnožena brojem čiji se faktorijel traži u prethodnom pozivu funkcije, daje vrednost tog faktorijela. Poslednja kreirana lokalna promenljiva w (prema pozivu faktr(0)) prva prestaje da postoji, a prva kreirana – prema pozivu faktr(5) poslednja prestaje da postoji.

Rekurzijom se obično ne štedi memorija jer je potreban prostor na stek memoriji za pamćenje podataka pri rekurzivnim pozivima funkcije. Iz stek memorije dodeljuje se prostor automatskim (lokalnim) promenljivim svih funkcija (ne samo rekurzivnih) pri svakom njihovom pozivu, kao i parametrima funkcija. Stek memorija je područje memorije iz kojeg se memorijski prostor dodeljuje po LIFO (Last In First Out) principu, tj. poslednje dodeljene lokacije se prve oslobađaju. Prostor se dodeljuje i oslobađa na vrhu (sa vrha) steka (više o strukturi steka v. u poglavlju o dinamičkim strukturama podataka).

Kada program počne da se izvršava od funkcije main(), rezerviše se stek okvir te funkcije kao prostor na steku za sve promenljive deklarisane unutar main(). Ako main() poziva neku funkciju, dodatni prostor se dodeljuje za parametre i automatske promenljive te funkcije na vrhu steka (stek okvir te funkcije). Ako ova funkcija poziva drugu funkciju (ili rekurzivno samu sebe), prostor (stek okvir) će se dodeliti na novom vrhu steka. Po povratku iz funkcije, prostor za njen stek okvir se oslobađa i vrh steka se vraća na prethodnu poziciju. Dakle, prostor na steku se zauzima, oslobađa i ponovo zauzima za vreme izvršavanja programa. To je privremeni prostor.

Rekurzijom se obično ne povećava ni brzina izvršavanja funkcije, ali je rekurzivni zapis kompaktniji, lakši za pisanje i razumevanje. Rekurzija je posebno pogodna u obradi rekurzivno definisanih struktura podataka kakva su drveta.

6.1 Primeri

6.1.1 Štampanje cifara celog broja

Jedno rešenje je računanje cifara kao ostataka pri deljenju broja sa 10, od cifre najmanje do cifre najveće težine, njihovo pamćenje u nizu, a zatim štampanje elemenata niza u obrnutom poretku. Druga mogućnost je rekurzivno rešenje u kome funkcija prvo poziva samu sebe da obradi (odredi i odštampa) vodeće cifre a zatim štampa poslednju cifru.

```
#include <stdio.h>
/* printd: stampa cifre celog broja n */
void printd(int n)
{
    if(n<0) {
        putchar('-');
        n = -n;
    }
    if(n/10)
        printd(n/10);
    putchar(n%10 + '0');
}
```

6.1.2 Hanojske kule

Zadatak: Data su tri štapa, i na jednom od njih n diskova različitih prečnika, poređanih tako da manji disk leži na većem. Potrebno je premestiti sve diskove na treći štap u istom poretku, tako što se premešta jedan po jedan disk, korišćenjem sva tri štapa, i u svakom trenutku na svakom od štapova disk može da leži samo na većem disku.

```
#include <stdio.h>

/* prebacuje n diskova sa i-tog na j-ti štap */
void prebaci (int n, int i, int j);

void main()
{
    int broj; /*broj diskova */
    printf("Unesite broj diskova");
    scanf("%d", &broj);
    prebaci(broj, 1, 3);
    return;
}

void prebaci(int n, int i, int j)
{
```

```
        int k;
    /* odrediti pomoćni štap k */
        switch(i+j) {
            case 3: k=3; break;
            case 4: k=2; break;
            case 5: k=1; break;
        }
        if(n==1)
            printf("Prebaceno sa %d na %d", i, j);
        else {
            prebaci(n-1, i, k);
            prebaci(1,i,j);
            prebaci(n-1, k,j);
        }
        return;
    }
```

Broj premeštanja n diskova je 2^n-1, pa je složenost algoritma eksponencijalna ($O(2^n)$).

6.1.3 Binarno pretraživanje

Iterativna funkcija za binarno pretraživanje sortiranog niza ima sledeću definiciju:

```
    int binsearch(int x, int b[], int m)
    {
        int l=0, d=m-1, s;
        while(l<=d){
            s=(l+d)/2;
            if(x<b[s]) d=s-1;
            else if(x>b[s]) l=s+1;
            else retrurn s;
        }
        return-1;
    }
```

Rekurzivna funkcija za binarno pretraživanje sortiranog niza ima sledeću definiciju:

```
    int binsearchr(int x, int b[], int l, int d)
    {
        int s;
        if(l>=d) return -1;
        else{
            s=(l+d)/2;
            if(x==b[s]) return s;
            else if(x<b[s]) d=s-1;
            else l=s+1;
```

```
            return binsearchr(x,b,l,d);
        }
    }
```

6.2 Eliminacija repne rekurzije

U slučaju da funkcija ima samo jedan rekurzivni poziv, postoji strogo utvrđen, korektan postupak za transformaciju funkcije u ekvivalentnu iterativnu funkciju. Taj postupak uključuje primenu GOTO iskaza, ali to je jedna od kontrolisanih upotreba ovog inače nepoželjnog iskaza, čija je korektnost i ponašanje strogo dokazano.

Razlikuju se dva slučaja ovakvih funkcija: kada je poslednji iskaz funkcije rekurzivni poziv te iste funkcije (tzv. repna rekurzija), i kada je jedini rekurzivni poziv funkcije unutar definicije funkcije. Razmotrićemo samo prvi slučaj.

Ako je poslednji iskaz funkcije P, pre povratka, poziv funkcije Q, taj poziv se može zameniti GOTO iskazom na početak funkcije Q. Ako je P≡Q, onda se GOTO iskaz usmerava na početak funkcije P.

Razmotrimo ponovo algoritam binarnog pretraživanja. Rekurzivna funkcija može imati i sledeći oblik:

```
int binarpretr(int x, int l, int d)
{
int s, bp;
      if (d< l) return -1;
      else {
                  s=(l+d) / 2;
                  if (x == b[s]) return s;
                  else
                        if (x < b[s])
                              {bp=binarpretr(x,l,s-1);
                              return bp; }
                        else
                              {bp=binarpretr(x,s+1,d);
                              return bp; }
            }
}
```

Ova funkcija ima samo jedan rekurzivni poziv (izvršava se ili iskaz bp = binarpretr(x,l,s-1) ili iskaz bp = binarpretr(x,s+1,d). Definicija ove funkcije se prvo može preformulisati tako da i u zapisu funkcije figuriše samo jedan rekurzivni poziv:

```
int binarpretr(int x, int l, int d)
{
  int s, bp;
    if (d< l) return -1;
    else {
            s=(l+d) / 2;
            if (x == b[s]) return s;
            else {
                if (x < b[s]) d = s-1;
                else l=s+1;
                bp=binarpretr(x,l,d);
                return bp;
                }
          }
}
```

Sada se funkcija može transformisati tako da se eliminiše repna rekurzija:

```
int binarpretr(int x, int l, int d)
{
  int s;
label:    if (d< l) return -1;
    else {
            s=(l+d) / 2;
            if (x == b[s]) return s;
            else {
                if (x < b[s]) d = s-1;
                else l=s+1;
                goto label;
                }
          }
}
```

Da bi se eliminisao GOTO iskaz iz prethodne deklaracije funkcije, potrebno je grupisati iskaze koji su u petlji, i iskaze kojima funkcija dobija vrednost i završava izvršavanje (return ...):

```
int binarpretr(int x, int l, int d)
{
  int s;
label:    if (d>= l)
          {
               s=(l+d) / 2;
               if (x ! = b[s])
               {
                    if (x < b[s]) d = s-1;
                    else l=s+1;
                    goto label;
               }
               else return s;          }
     else return -1;
}
```

Sada se deo definicije funkcije binarpretr, koji se ponavlja pomoću GOTO iskaza, može zameniti eksplicitnim iskazom petlje, tj. definicija funkcije dobija oblik:

```
int binarpretr(int x, int l, int d)
{
  int s;
     while (d>= l)
     {
          s=(l+d) / 2;
          if (x != b[s])
          {
               if(x < b[s]) d = s-1;
               else l = s+1;
          }
          else return s;
     }
     return -1;
}
```

7

Rekurzivni algoritmi sortiranja

7.1 Brzi sort (Quicksort)

Ovo je najčešće upotrebljavan algoritam sortiranja. Osnovni oblik algoritma dao je 1960, Hor (Hoare). Nije težak za implementaciju, a koristi manje resursa (vremena i prostora) nego bilo koji drugi algoritam sortiranja, u većini slučajeva. Algoritam ne zahteva dodatnu memoriju, samo $n\log n$ operacija u proseku za sortiranje n elemenata, i ima izuzetno kratku unutrašnju petlju. Loše strane algoritma su što je rekurzivan (nerekurzivna varijanta je mnogo složenija), u najgorem slučaju izvršava oko n^2 operacija. Postoje i verzije ovog algoritma koje ga poboljšavaju.

Algoritam je vrlo osetljiv na implementaciju (efikasnost se može narušiti lošim izborom u implementaciji). Ako se ne želi analizirati najbolja implementacija, bolje je primeniti šelsort.

Ideja algoritma sastoji se u particioniranju niza prema odabranom elementu particioniranja koji se dovodi na pravo mesto, i u primeni algoritma brzog sortiranja na svaku od dve dobijene particije. Rekurzivni poziv se završava kada se primeni na particiju sa manje od dva elementa. Argumenti funkcije, pored niza koji se sortira, još su i indeks krajnjeg levog i krajnjeg desnog elementa niza (l, d). Osnovni algoritam ima sledeći oblik:

```
void quicksort (int x[ ], int l, int d)
{
      int i;
      if (d > l) {
            i=partition(l,d);
            quicksort(x, l, i–1);
            quicksort(x, i+1, d);
      }
}
```

Argumenti l,d ograničavaju podniz koji će biti sortiran. Poziv quicksort(x, 1,n) sortira ceo niz. Funkcija partition preuređuje niz tako da su zadovoljeni uslovi:

(i) $x[i]$ je na svom mestu za neko i,

(ii) svi elementi $x[l], \ldots, x[i-1]$ su manji ili jednaki sa $x[i]$,

(iii) svi elementi $x[i+1], \ldots, x[d]$ su veći ili jednaki sa $x[i]$.

Ovo može biti realizovano sledećom strategijom: Prvo, proizvoljno se odabere $x[d]$ kao element koji će ići na svoju konačnu poziciju. Zatim se prolazi kroz niz od njegovog levog kraja dok se ne nađe element veći od $x[d]$ i prolazi se od desnog kraja niza dok se ne nađe element manji od $x[d]$. Dva elementa koja su zaustavila ova dva prolaza su na pogrešnom mestu, i njih razmenjujemo. Nastavljanjem ovog postupka dobija se da su svi elementi levo od levog markera manji od (ili jednaki sa) $x[d]$, i da su svi elementi desno od desnog markera veći od (ili jednaki sa) $x[d]$. Kada se ovi markeri sretnu,treba još samo razmeniti $x[d]$ sa elementom na kome su se markeri sreli, i proces particioniranja je gotov.

Primer za particioniranje (podvučen je element particioniranja):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
P	R	I	M	E	R	Z	A	S	O	R	T	I	R	A	NJ	<u>E</u>
A	R	I	M	E	R	Z	A	S	O	R	T	I	R	P	NJ	<u>E</u>
A	A	I	M	E	R	Z	R	S	O	R	T	I	R	P	NJ	<u>E</u>
A	A	<u>E</u>	M	E	R	Z	R	S	O	R	T	I	R	P	NJ	I

Krajnji desni element, E, izabran je kao element za particioniranje. Prvo se prolaz sleva zaustavlja kod P, zatim se prolaz zdesna zaustavlja kod prvog zdesna A, zatim se ta dva elementa razmene (drugi red). Zatim se levi prolaz zaustavlja kod R a desni kod A i oni se razmene (treći red). Zatim se markeri poklapaju kod I, krajnje desno E se zamenjuje sa I i tako se dobije particionirani niz u četvrtom redu. Sortiranje se završava sortiranjem dva podniza na obe strane elementa particioniranja (rekurzivno).

Za jednu punu implementaciju funkcije quicksort v. Kernighan, Ritchie, "The C Programming Language".

Quicksort metoda pokazuje vrlo loše karakteristike kada se primenjuje na već sortirani niz (oko $n^2/2$ operacija i n lokacija za obradu rekurzije). Najbolji slučaj koji bi mogao da se dogodi je da se u svakoj fazi particioniranja niz podeli tačno na pola. To bi omogućilo da broj poređenja u quicksort-u zadovolji rekurentnu vezu "podeli-pa-vladaj" mehanizma:

$C(n) = 2 * C(n/2) + n$ (uz $C(1) = 1$).

($2 * C(n/2)$ predstavlja vreme obrade dva podniza; n je vreme ispitivanja svakog elementa, upotrebom jednog ili drugog markera particioniranja).

Da bi se procenila vrednost $C(n)$, izaberu se vrednosti za n oblika 2^N:

$C(2^N) = 2 * C(2^{N-1}) + 2^N \qquad /2^N$

$\frac{C(2^N)}{2^N} = \frac{C(2^{N-1})}{2^{N-1}} + 1$

Primenjujući istu formulu N puta dobije se samo N kopija "1", tj. $C(2^N) = N * 2^N$. Kako to važi za skoro svako n, dobija se formula $C(n) \sim n \ln n$. (Ako bi

algoritam bio takav da se posle polovljenja niza jedna polovina ne mora obrađivati, već samo jedna – rekurzivno, dobilo bi se vreme $C(n) = C(n/2) + 1$, odakle se odmah dobija da je $C(2^N) = N$, i $C(n) \sim \ln n$).

Ovo je najbolji slučaj ali se može dokazati da je i srednji slučaj proporcionalan sa $n \ln n$. (Ovde se neće razmatrati poboljšanja kao što je eliminacija rekurzije, slučajni izbor elementa particioniranja (ili aritmetička sredina tri slučajna elementa za element particioniranja itd.)

7.2 Jedna implementacija

Jedna implementacija algoritma quicksort, predstavljena u "The Programming Language C" B. Karnighan, D. Ritchie, je jednostavna mada ne i najefikasnija. Za element particionisanja uzima središnji element i na početku ga "sklanja" na krajnju levu poziciju (tj. razmenjuje ga sa elementom na krajnjoj levoj poziciji) . Zatim prolazi kroz sve ostale elemente (sleva na desno), i svaki koji je manji od elementa particionisanja premešta na levi kraj niza. Promenljiva "poslednji" pamti najveći indeks elementa manjeg od elementa particionisanja. Kada se prođe kroz sve elemente niza, element particionisanja se razmenjuje sa elementom na poziciji poslednji (to je njegova konačna pozicija u sortiranom nizu), a zatim se funkcija qsort rekurzivno primenjuje na levi i desni podniz (u odnosu na poziciju poslednji).

```
void qsort(int v[], int l, int d)
{
    int i, poslednji;
    void swap(int v[], int i, int j);
    if(l>= d) return;
    swap(v,l,(l+d)/2);
    poslednji=l;
    for(i=l+1; i<=d; i++)
        if (v[i]< v[l]) swap(v, ++poslednji, i);
    swap(v, l poslednji);
    qsort(v, l , poslednji-1);
    qsort(v, poslednji+1, d);
}
```

7.3 Sortiranje spajanjem (Merge sort)

Sortiranje spajanjem ili "Merge sort" je algoritam sortiranja zasnovan na poređenju. To je rekurzivni algoritam. Njegova vremenska složenost proporcionalna je sa $n\log n$, a (dodatna) prostorna složenost je proporcionalna sa n (postoji i implementacija u kojoj je proporcionalna sa $\log n$). U većini implementacija je stabilan, što znači da zadržava početni redosled jednakih elemenata u sortiranom nizu. Predstavlja primer algoritamske paradigme "podeli pa vladaj". Konstruisao ga je Džon fon Nojman (John von Neumann) 1945. godine.

Konceptualno, algoritam sortiranja spajanjem "radi" na sledeći način:

- Ako niz ima nula ili jedan element, onda je već soritran. Inače,
  - Podeliti nesortirani niz u dva podniza približno jednake dužine.
  - Sortirati svaki podniz rekurzivno ponovnom primenom algoritma sortiranja spajanjem.
  - Spojiti dva sortirana podniza u jedan sortirani niz.

Algoritam sortiranja spajanjem uključuje dva važna principa kojima poboljšava (smanjuje) vreme izvršavanja:

1. kratki niz je moguće sortirati u manjem broju koraka nego dugački (osnova za deljenje niza na dva podniza)
2. manje koraka je potrebno za konstrukciju sortiranog niza od dva sortirana podniza (jedan prolaz kroz svaki od podnizova) nego od dva nesortirana podniza (osnova za spajanje).

Ako je niz koji se sortira označen sa A i ima n elemenata sa indeksima od 0 do n-1, onda se on podeli na podnizove *levi* (sa elementima $A_0, \ldots, A_{c-1}$) i *desni* (sa elementima $A_c, \ldots, A_{n-1}$) gde je c celobrojni deo količnika $n/2$. Te dve polovine se sortiraju a zatim spoje u sortirani niz A. Umesto spajanja moguće je primeniti jednostavno dopisivanje elemenata podniza *desni* na podniz *levi*, ukoliko je najveći element podniza *levi* manji ili jednak najmanjem elementu podniza *desni* (moguće je i obratno).

Algoritam bi mogao da se opiše na sledeći način (nije programski zapis vec tzv. pseudoprogramski zapis algoritma):

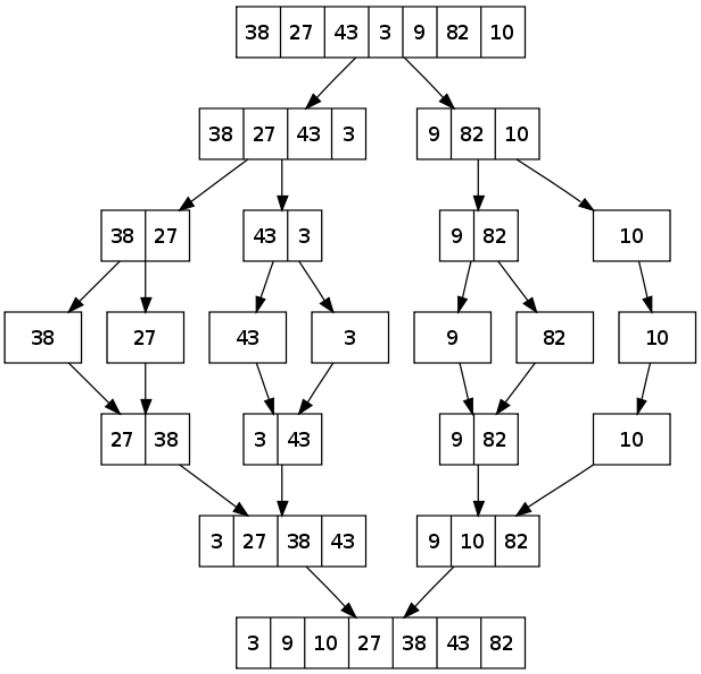
```
algoritam merge_sort(A)
{
    tip_elementa_niza levi[ ], desni[ ], rezultat[ ];
    int sredina;
    if (length(A) <= 1)
        return A;
    sredina = length(A) / 2;
    for (i=0; i<=sredina; sredina++)
        upisati A[i] u niz levi;
    for (i=sredina+1; i<= length(A)-1; sredina++)
        upisati A[i] u niz desni;
    levi = merge_sort(levi);
    desni = merge_sort(desni);
    if levi[sredina] > desni[0]
        rezultat = spoji(levi, desni);
    else
        rezultat = dodaj(levi, desni;)
```

```
    return rezultat;
}
```

Funkcija spoji mogla bi da se opiše sledećim algoritmom:

```
function spoji(levi,desni)
{
/* niz levi ima length(levi) elemenata: levi[0..length(levi)-1] */
/* niz desni ima length(desni) elemenata: desni[0..length(desni)-1] */
/* niz rezultat imaće length(levi)+length(desni) elemenata */
    tip_elementa_niza rezultat[ ];
    while length(levi) > 0 && length(desni) >0
        if levi[0] <= desni[0]
        {
            upiši levi[0] u niz rezultat;
    /* iz niza levi isključuje se prvi element - onaj sa indeksom 0 */
            levi = levi[1..length(levi)-1];
        }
        else
        {
            upiši desni[0] u niz rezultat;
    /* iz niza desni isključuje se prvi element - onaj sa indeksom 0 */
            desni = desni[1..length(desni)-1];
        }
    if length(levi) > 0
        upiši ostatak niza levi u niz rezultat;
    else
        upiši ostatak niza desni u niz rezultat;
    return rezultat;
}
```

Primer: Sortiranje niza celih brojeva algoritmom spajanja (Merge sort)


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

8

Pokazivači – napredne teme

8.1 Pokazivači i nizovi

Već smo naglasili, govoreći o nizovima u C-u, da su pokazivači i nizovi tako tesno povezani da zaslužuju istovremeni prikaz. Sve što može da se uradi korišćenjem indeksiranih elemenata niza, može i korišćenjem pokazivača.

Na primer, deklaracija

int a[10];

definiše niz od 10 elemenata, tj. rezerviše 10 susednih objekata nazvanih a[0], a[1], . . . , a[9].

Oznaka a[i] označava i-ti element niza. Ako je pa pokazivač na ceo broj, deklarisan sa

int ∗pa;

tada iskaz dodele

pa = &a[0];

dodeljuje promenljivoj pa pokazivač na nulti element niza a, tj. pa sadrži adresu elementa a[0].

Grafički (slika 8.1):

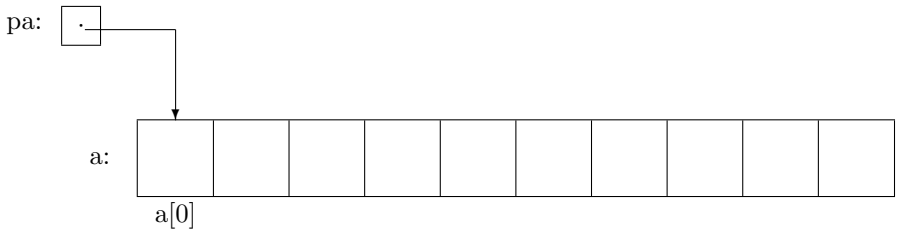


Figure 8.1: Pokazivač na niz

Sada dodela

x = ∗pa;
kopira sadržaj elementa a[0] u promenljivu x.

Ako pokazivač pa pokazuje na neki određeni element niza, onda, po definiciji, pa+1 pokazuje na sledeći element a pa-1 na prethodni element niza. Ako pa pokazuje na element a[0], onda je pa+i – *adresa* elementa a[i] a ∗(pa+i) je *sadržaj* elementa a[i]. Ova aritmetika nad pokazivačima (adresama) važi bez obzira na tip objekta – elementa niza.

Ime niza **a** tj. izraz **a** ima tip int [10] a vrednost adrese njegovog početnog elementa; zato dodela
pa = &a[0]; (pokazivač dobija adresu nultog elementa niza),
ima ekvivalentno značenje kao i dodela
pa = a;
a tip int [10]se može konvertovati u tip int ∗. Zato se funkciji koja kao argument ima ime niza elemenata tipa int (npr. f(int a[])), može proslediti i pokazivač na objekat odgovarajućeg tipa (npr. f(int ∗)).

Šta više, a+i (adresa i-tog elementa) je isto što i &a[i], pa je i ∗(a+i) jednak a[i], i to je upravo način na koji se pristupa sadržaju elementa a[i] u C-u. Slično, ako je pa pokazivač, onda je ∗(pa+i) isto što i pa[i].

Ipak, bitna razlika između pokazivača i imena niza je u tome što je pokazivač – promenljiva, pa ima smisla izraz pa=a ili pa++, dok ime niza to nije, pa nema smisla izraz oblika a=pa ili a++. Dakle, pokazivač pa jeste l-vrednost, dok ime niza a to nije.

8.2 Alokacija memorije; nizovi pokazivača i višedimenzioni nizovi

Definicijom
char a[10][100];
definiše se dvodimenzioni niz karaktera "a" kao niz od 10 jednodimenzionih nizova sa po 100 elemenata tipa char. Dakle, svaki element tog jednodimenzionog niza – a[0], a[1], . . . , a[9] – ima tip char [100] i predstavlja pokazivač na svoj početni element – a[0] na a[0][0], a[1] na a[1][0], . . . , a[9] na a[9][0]. Pri tome je a+i, kao i za druge nizove, isto što i a[i], dok je a[0]+i isto što i &a[0][i].

Kako je svako ime niza – pokazivač na njegov početni element, a niz "a" je niz od 10 elemenata – jednodimenzionih nizova sa po 100 elemenata tipa char, ime niza "a" se konvertuje u pokazivač na prvi od tih jednodimenzionih nizova, tj. u pokazivač
char (∗a)[100];
Tako su i a i a[0] – pokazivači na početni element dvodimenzionog niza.

Dvodimenzioni niz i niz pokazivača imaju puno sličnosti. Na primer, definicijom
char ∗b[10];
definiše se i niz pokazivača na karaktere (ili prve karaktere stringova) "b".

Pri tome, obe oznake a[3][4] i b[3][4] predstavljaju ispravna obraćanja jednom karakteru.

Međutim, bitna razlika u definisanju dvodimenzionog niza i niza pokazivača na nizove, jeste u alokaciji (dodeljivanju) memorijskog prostora. Tako, prvom definicijom se rezerviše prostor za 1000 znakova (karaktera) a elementu (karakteru) a[vrsta][kolona] dodeljen je prostor sa indeksom 100×vrsta + kolona unutar tih 1000 lokacija. S druge strane, definicija niza b rezerviše prostor **samo** za 10 pokazivača, i ne inicijalizuje te pokazivače (ne dodeljuje im nikakve vrednosti tj. adrese memorijskih prostora); obezbeđenje memorijskih lokacija na koje će ti pokazivači da pokazuju, i dodela adresa tih lokacija pokazivačima, mora da se obavi eksplicitno, npr. funkcijom malloc, calloc i sl. Ipak, bitna prednost nizova pokazivača u odnosu na dvodimenzioni niz je u tome što prostori na koje pokazuju pojedini elementi niza pokazivača mogu da budu različite dužine. Dakle, ne mora svaki element niza b da pokazuje na string dužine 100 karaktera.

Na primer,
b[i] = (char *)malloc(strlen(a[i])+1); (pokazivač b[i] pokazuje na prostor veličine stringa u i-toj vrsti matrice a)
b[i] = (char *)malloc(sizeof(*b[i])); (pokazivač b[i] pokazuje na prostor veličine jednog karaktera)
b[i] = (char *)malloc(10*sizeof(*b[i])); (pokazivač b[i] pokazuje na prostor veličine 10 karaktera – bajtova)

Operator sizeof je unarni operator koji se izvršava u vreme kompilacije i primenjuje se na objekat ili tip: sizeof(object), sizeof(type-name). Izračunava veličinu objekta (ili vrednosti navedenog tipa) u bajtovima i vraća tip size_t – unsigned integer definisan u standardnom zaglavlju <stddef.h>. Kada se ovaj operator primeni na ime niza, izračunava ukupni broj bajtova koje zauzimaju svi elementi tog niza.

Funkcije malloc, calloc, realloc za dinamičku dodelu memorijskog prostora u vreme izvršavanja (i odgovarajuća funkcija za oslobađanje memorijskog prostora free) definisane su u standardnom zaglavlju <stdlib.h>:

void *malloc(size_t n)
vraća pokazivač na n bajtova neinicijalizovanog prostora ili NULL ako prostor ne može da se dodeli.

void *calloc(size_t n, size_t size)
vraća pokazivač na dovoljan prostor u memoriji za niz od n objekata navedene veličine size, ili NULL. Prostor je inicijalizovan sa 0.

void* realloc (void* ptr, size_t size);
menja veličinu prethodno dodeljenog memorijskog bloka na koji pokazuje pokazivač ptr, na veličinu "size" . Sadržaj bloka se čuva u prostoru čija je veličina jednaka manjoj između stare i nove veličine bloka. Dodatno alocirani prostor (ako je novi prostor veći od starog) ima nedefinisan sadržaj.

Povratni tip void * ukazuje da povratna vrednost (pokazivač) mora da bude pridružena odgovarajućem tipu, na primer:

```
int *ip;
ip=(int *)calloc(n, sizeof(int));
```

Slično važi za funkciju malloc().

Funkcija void free(void $*$) oslobađa memorijski prostor dobijen pozivom funkcije malloc() ili calloc(). Na primer, free(p) oslobađa prostor na koji pokazuje p, ako je p dobijen pozivom funkcije malloc ili calloc (free(ip) za gornji primer).

Ranije je bilo reči o jednom segmentu memorije, tzv. stek memoriji koja se koristi za smeštanje lokalnih promenljivih funkcija i njihovih argumenata, pri svakom pozivu funkcije. Takođe, bilo je reči i o segmentu podataka – segmentu memorije na kome se čuvaju statičke promenljive – globalne promenljive i lokalne statičke promenljive. Pomenimo ovde još jedan segment memorije iz kojeg se dinamički dodeljuje prostor pozivom funkcija malloc(), calloc(). To je tzv. heap memorija (hip). Za razliku od stek memorije koja se automatski dodeljuje i oslobađa pri pozivu funkcija i povratku iz funkcija, hip memorija se dodeljuje eksplicitnim pozivom funkcija za dinamičko dodeljivanje i oslobađanje memorije (malloc(), calloc(), free()).

8.3 Pokazivači na funkcije

U programskom jeziku C funkcija nije promenljiva, ali je moguće definisati pokazivače na funkcije, kojima se mogu dodeliti vrednosti, koji se mogu smestiti u niz, proslediti funkcijama kao argumenti, koji se mogu vratiti kao vrednosti funkcija, itd. Ilustrujmo pokazivače na funkcije primerom jedne funkcije koja kao argument koristi drugu funkciju, i to tako što se prvoj funkciji predaje, kao argument, pokazivač na drugu funkciju.

Primer 1 – funkcija kao argument: Napisati program koji izračunava približnu nulu funkcije f (sa zadatom tačnošću) na intervalu [a,b] (na kome se zna da funkcija f ima tačno jednu nulu) primenom metode polovljenja intervala.

Rešenje se može realizovati tako što se metoda polovljenja intervala realizuje funkcijom "nula" (prva funkcija) koja ima argumente – pokazivač na funkciju f (druga funkcija) za koju se traži nula i granice intervala a,b, a zatim se u programu (u glavnoj funkcji) poziva funkcija "nula" sa stvarnim argumentima – funkcijom za koju se traži nula i granicama intervala.

Pretpostavka metode polovljenja intervala je da funkcija čija se približna nula traži ima tačno jednu nulu na zadatom intervalu, tj. da funkcija ima različit znak na krajevima intervala. Metoda se sastoji u polovljenju intervala na kome se nula traži, i to tako da je znak funkcije u graničnim tačkama intervala različit. Tako, ako je znak funkcije u tačkama a i b različit, i ako je znak funkcije u tački a isti kao znak funkcije u sredini intervala $[a,b]$, onda se sredina intervala $[a,b]$ proglašava za novu tačku a – tj. levu granicu novog, prepolovljenog intervala; u suprotnom, sredina intervala $[a,b]$ proglašava se za novu tačku b. Postupak se ponavlja sve dok

interval [*a*,*b*] ne postane dovoljno mali (manji od unapred zadatog malog broja), ili dok vrednost funkcije u sredini intervala ne postane dovoljno mala; za nulu funkcije proglašava se sredina poslednjeg intervala.

```
#include <math.h>
#include <stdio.h>
#define EPS 0.0001
float nula(float a, float b, float (*f)(float));
float f1(float);
float f2(float);
main()
{
    float x,y;
    scanf("%f, %f", &x, &y);
    printf("%f, %f, %f\n", x,y, nula(x,y,f1));
    scanf("%f, %f", &x, &y);
    printf("%f, %f, %f\n", x,y, nula(x,y,f2));
}
float nula(float a, float b, float (*f)(float))
{
    float x,z;
    int s;
    s=((*f)(a)<0);
    do {
        x=(a+b)/2.0;
        z=(*f)(x);
        if((z<0)==s) a=x; else b=x;
    } while (fabs(z)>EPS);
    return x;
}
float f1(float x)
{
    float y;
    y= x–1;
    return y;
}
float f2(float x)
{
    float y;
    y= x*x–1;
    return y;
}
```

U pozivu funkcije nula, f1 i f2 su adrese funkcija f1 odnosno f2 (jer je ime funkcije, kao i ime niza, zapravo adresa funkcije tj. niza). Funkcije f1 i f2 su

jednostavne funkcije – linearna i kvadratna. Nula se u ovom primeru računa sa tačnošću od 10^{-4}.

Primer 2 – tabeliranje funkcije Treba napisati funkciju `tabeliranje` koja za proizvoljnu zadatu funkciju izračunava i štampa njene vrednosti u zadatom intervalu sa zadatim korakom. Napisati i program koji pozivom funkcije `tabeliranje` prikazuje vrednosti funkcije $f(x) = x^2 - 1$ na intervalu $[1, 2]$ sa korakom 0.01.

```
#include <stdio.h>
void tabeliranje(float (*f)(float), float, float, float);
float f1(float);
int main()
{
    tabeliranje(f1, 1.0,2.0,0.01);
}

float f1(float x)
{
    return x*x-1;
}

void tabeliranje(float (*f)(float), float a, float b, float h)
{
    float x=a;
    while(x<=b)
    {
        printf("%f, %f \n", x, (*f)(x));
        x+=h;
    }
}
```

8.4 Standardne funkcije bsearch i qsort

Do sada implemetirane funkcije za pretraživanje i sortiranje nizova mogle su da se primene samo na nizove sa unapred određenim tipom elemenata (npr. int). Zadati tip elemenata niza određivao je memorijski prostor potreban za registrovanje jednog elementa, pa i preciznu aritmetiku nad pokazivačima kojima se tim elementima pristupa: adresa i+1. elementa jednaka je adresi i-tog elementa uvećanoj za prostor potreban za njegovo registrovanje. Takođe, za zadati tip elemenata niza, poznat je način na koji se definiše (i proverava) uređenje (na primer, za celobrojne elemente to je operacija poređenja <, za strigove to je funkcija strcmp, itd).

U standardnoj C-biblioteci postoje (rekurzivne) funkcije za binarno pretraživanje sortiranog niza (bsearch) i za brzo sortranje niza (qsort) sa elementima proizvoljnog tipa. Za njih je neophodno precizirati (kao argumente) veličinu memorijskog prostora potrebnog za jedan element kao i funkciju kojom se vrši

upoređivanje elemenata. Ova funkcija se prenosi – kao argument – funkcijama bsearch odnosno qsort mehanizmom pokazivača na funkcije.

Deklaracije funkcija qsort i bsearch nalaze se u standardnom zaglavlju <stdlib.h> i imaju sledeći oblik:

void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *),

gde je base - ime niza koji se sortira, n – broj elemenata niza, size – veličina elementa niza (u bajtovima), cmp – pokazivač na funkciju koja vraća int: 0 ako su dva argumenta – elementa niza jednaka, vrednost manju od 0 ako je vrednost prvog argumenta manja od druge, i vrednost veću od 0 ako je vrednost prvog argumenta veća od druge. "const void *" znači da funkcija za poređenje ne sme da menja svoje argumente, koji su inače pokazivači na proizvoljni tip (tip elementa niza). Funkcija qsort, kao i ranije funkcije za sortiranje, ne vraća vrednost.

Slično,

void * bsearch (const void * key, const void *base, size_t n, size_t size, int (*cmp)(const void * , const void *)

gde su base, n i size kao i za funkciju qsort, key je pokazivač na vrednost koja se traži u nizu base. Funkcija bsearch poziva funkciju cmp za poređenje vrednosti ključa (key) sa pojedinačnim elementima niza base. Funkcija cmp u ovom slučaju ima oblik

int cmp (const void * key, const void * pelem);

Prvi argument je pokazivač na vrednost koja se traži, a drugi argument je pokazivač na element niza base. Tipovi ovih argumenata ne moraju biti isti, a način poređenja definiše funkcija koja realizuje cmp. Na primer, vrednost koja se traži može biti string, a element niza može biti struktura sa dva člana od kojih je prvi tipa string. Tada se poređenje realizuje kao poređenje prvog argumenta sa vrednošću člana tipa string drugog argumenta.

Funkcija bsearch vraća pokazivač na tip za koji je vezana pokazivačka promenljiva koja prima vraćenu vrednost.

Primer 1. Sortiranje celobrojnog niza

```
#define NMAX 1000;
int v[NMAX];
int cmpint(const int *, const int *);
...
qsort(v, NMAX, sizeof(int), cmpint);
...
int cmpint(const int * x, const int * y)
{
     return *x−*y;
```

```
}
```

Primer 2. Sortiranje niza karaktera

```
#define MAXCH 1000;
char ch[MAXCH];
int chcmp(const char *, const char *);
...
qsort(ch, n, sizeof(char), chcmp);
...
int chcmp(const char * c1, const char * c2)
{
    return *c1−*c2;
}
```

Primer 3. Sortiranje niza pokazivača na stringove

```
char *reci[]={"auto", "while", "break", ...};
int cmpreci(const char **, const char **);
...
qsort(reci, NREC, sizeof(char *), cmpreci);
...
int cmpreci(const char ** s1, const char ** s2)
{
    return strcmp(*s1, *s2);
}
```

Primer 4. Sortiranje niza struktura

```
struct kljuc {
    char * rec;
    int brojac
} kljucreci[] = {
        "auto", 0,
        "while", 0,
        "break", 0,
        ...};
int cmpstruct(const struct kljuc *, const struct kljuc *);
...
qsort(kljucreci, NKLJUC, sizeof(struct kljuc), cmpstruct);
...
int cmpstruct(const struct kljuc *s1, const struct kljuc *s2)
{
    return strcmp(s1→rec, s2→rec);
}
```

Primer 5. Pretraživanje niza celih brojeva

```
int *p, *broj;
p = (int *)bsearch(broj, nizbrojeva, n, sizeof(int), cmpint);
(cmpint kao u primeru 1.)
```

Primer 6. Pretraživanje niza struktura

```
struct kljuc {
    char * rec;
    int brojac
} *p, tabelak[] = {
        "auto", 0,
        "break", 0,
        "while", 0,
... };
p = (struct kljuc *)bsearch(rec, tabelak, NKLJUC, sizeof(struct kljuc),
strtcmp);
int strtcmp(const char * rec, const struct kljuc * entry)
{
    return strcmp(rec, entry → rec);
}
```

8.5 Argument lista promenljive dužine

Videli smo ranije da funkcija printf, scanf i njima srodne funkcije, mogu da budu pozivane sa različitim brojem argumenata, tj. da imaju argument listu promenljive dužine. Na primer,

```
printf("%d, %d \n", i, j);      (3 argumenta)   ili
printf("%f \n", x);             (2 argumenta)   ili
printf("ovo je poruka \n");     (1 argument)
```

Funkcija printf zapravo je deklarisana sa

```
int printf(char *format, ... )
```

gde je jedini imenovani argument funkcije printf - format, a deklaracija "..." znači da broj i tip tih argumenata može da varira. Deklaracija "..." može da se pojavi samo na kraju argument liste.

Da bi se napisala definicija funkcije sa promenljivim brojem argumenata, potrebno je da se koriste neke makro definicije iz standardnog zaglavlja <stdarg.h>, koje će da omoguće prolaz kroz argument listu promenljive dužine. To su:

- va_list – tip promenljive koja će redom da pokazuje na jedan po jedan argument argument liste ("va" - od variable argument- promenljivi argumenti). Na primer,

  va_list ap; (ap za argument pointer)

- va_start – makro koji inicijalizuje promenljivu ap tako da pokazuje na prvi neimenovani argument (prvi posle argumenta format u funkciji printf); mora da se pozove jednom pre nego što ap može da se koristi. Mora da postoji bar jedan imenovani argument (npr. format), a va_start polazi od poslednjeg imenovanog argumenta jer iza njega kreću neimenovani). Na primer,

  va_start(ap, format);

- va_arg – makro koji vraća tekući argument i pomera ap na sledeći. Kao argument, pored ap, ima i naziv tipa koji treba da vrati. Na primer,

  va_arg(ap, int);

- va_end – makro koji završava rad sa promenljivom argument listom. Mora da se pozove pre završetka funkcije. Na primer,

  va_end(ap);

Da bismo pokazali kako se implementira funkcija sa promenljivim brojem argumenata, prikazaćemo implementaciju minimalne verzije funkcije printf, nazvane minprintf (implementacija iz Ritchie, Kernighan: The C Programming Language).

Funkcija minprintf takođe je deklarisana sa promenljivim brojem argumenata ali bez povratne vrednosti, tj. void minprintf (char *format, ...)

U svojoj implementaciji minprintf koristi printf za formatiranje, ali printf sa tačno dva argumenta, tj. samo jednim argumentom iza formata, dakle kao funkciju sa argument listom fiksne dužine.

```
#include <stdio.h>
#include <stdarg.h>
/* minprintf: minimalna printf funkcija sa promenljivom argument listom */
void minprintf(char *fmt, ...)
{
    va_list ap; /* pokazuje, redom, na svaki neimenovani argument */
    char *p, *sval;
    int ival;
    double dval;
    va_start(ap, fmt); /* postavlja ap da pokazuje na prvi neimenovani arg */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p)
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
```

```
                break;
            case 'f':
                dval = va_arg(ap, double);
                printf("%f", dval);
                break;
            case 's':
                for (sval = va_arg(ap, char *); *sval; sval++)
                    putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
            }
        }
        va_end(ap); /* "počistiti" na kraju */
    }
```

Primer 2. Funkcija average izračunava srednju vrednost svojih (double) argumenata, koliko god da ih ima. Prvi argument je upravo broj argumenata čiju srednju vrednost funkcija izračunava.

```
#include <stdarg.h>
#include <stdio.h>
// ova funkcija uzima prvo broj vrednosti čiji avg se računa,
// a zatim i same vrednosti čiji avg se računa
double average ( int broj, ... )
{
    va_list argumenti;
    double suma = 0;
    va_start ( argumenti, broj ); // inicijalizuje promenljivu argumenti
    for ( int x = 0; x < broj; x++ ) // petlja dok se ne saberu svi argumenti
        suma += va_arg ( argumenti, double ); // dodaje sledecu vrednost iz
argument liste na sumu
    va_end ( argumenti ); // brise listu
    return suma/broj; // vraca srednju vrednost
}

int main()
{
    // izracunava srednju vrednost od 13.2, 22.3 i 4.5 (3 oznacava broj vrednosti
cija se srednja vrednost racuna)
    printf("%f \n", average ( 3, 12.2, 22.3, 4.5 ));
    // ovde izracunava srednju vrednost 5 vrednosti 3.3, 2.2, 1.1, 5.5 i 3.3
    printf("%f \n", average ( 5, 3.3, 2.2, 1.1, 5.5, 3.3 ));
}
```

9

Dinamičke strukture podataka i algoritmi

9.1 Pokazivači i dinamičke strukture podataka

Već smo videli kako se pokazivači u C-u koriste za pokazivanje na promenljive koje su uvedene definicijom promenljivih, i čije je trajanje u skladu sa njihovom vrstom. Na primer, definicija

int ip, ∗p;

ako se nađe u okviru definicije neke funkcije, definiše celobrojnu promenljivu ip i pokazivač na celobrojnu promenljivu, ∗p, koje traju koliko i jedno izvršavanje te funkcije. Iskaz

p=&ip;

dodeljuje pokazivaču p adresu celobrojne promenljive ip.

S druge strane, pokazivaču p može da se dodeli adresa promenljive kreirane u toku izvršavanja jedne funkcije (za promenljivu se memorija alocira – dodeljuje nekom od funkcija calloc, malloc, itd. u toku izvršavanja te funkcije, npr. p=(int ∗)malloc(sizeof(∗p))), a promenljiva na koju pokazuje pokazivač p može da se ukloni (i da se oslobodi memorijski prostor) u toku izvršavanja te funkcije (npr. funkcijom free: void free(void∗ p).

Pošto ova alokacija/dealokacija memorije za promenljivu može da se izvršava u toku izvršavanja programa, ta promenljiva se naziva *dinamičkom promenljivom*. Ako su dinamičke promenljive – strukture sa nekim članovima – pokazivačima na strukture istog tipa, pomoću njih mogu da se kreiraju različite, kompleksne i fleksibilne, *dinamičke strukture podataka*. Za njih se ne mora unapred znati koliko elemenata (struktura) sadrže, i najčešće se i koriste za rešavanje problema u kojima nije unapred poznat broj elemenata (dimenzija problema).

Ako je tip T – tip strukture koja sadrži jedan ili više članova tipa T ∗ (pokazivač na tip T), onda se takvim tipom T mogu graditi strukture ekvivalentne

proizvoljnom konačnom grafu (v. sliku 9.22). Strukture tipa T tada predstavljaju čvorove grafa, a pokazivači – grane grafa.

9.2 Povezane liste

Lista je struktura koja se rekurzivno definiše kao konačni niz atoma ili lista. Atom je element nekog (proizvoljnog) skupa, i razlikuje se od liste. Ili,

(i) prazan skup atoma (čvorova) je lista;

(ii) ako je h – čvor a t – lista, onda je i uređeni par (h, t) – lista; h se naziva *glavom* liste (h, t), a t – *repom* liste (h, t).

Bekusovom notacijom lista se može zadati sledećim metalingvističkim formulama:

< lista > ::= < prazna lista > | (< glava>, < rep >)
< prazna lista > ::= ()
< glava > ::= element skupa
< rep > ::= < lista >

Lista je veoma opšta struktura; njom se može predstaviti, na primer, svaka drvoidna struktura (v. tačku 9.3.2). Na primer, aritmetički izraz $a - b(c/d + e/f)$ (tj. njegovo drvo, predstavljeno na slici 9.1), može se transformisati u drvo liste tog izraza (slika 9.2), a ono u samu listu (slika 9.3).

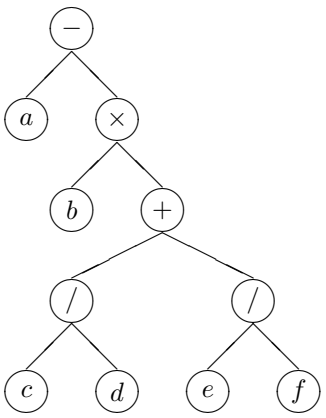


Figure 9.1: Drvo aritmetičkog izraza $a - b(c/d + e/f)$

Specijalni slučaj liste je linearna lista (tj. linearno drvo). To je skup koji se sastoji od $n \geq 0$ čvorova $x[1], x[2], \ldots x[n]$ čiji je položaj linearan, tj. za koji se zna koji je čvor prvi, koji je poslednji i koji za kojim sledi.

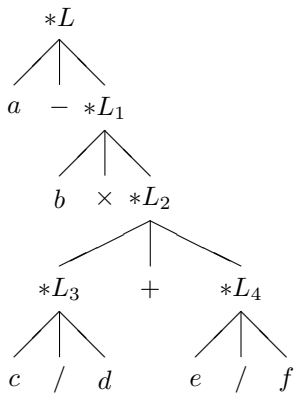


Figure 9.2: Drvo liste aritmetičkog izraza $a - b(c/d + e/f)$

$$L = (a, -, (b, x, ((c, /, d), +, (e, /, f))))$$

$$L[1] = a, \quad L[2] = -, \quad L[3] = (b, x, ((c, /, d), +, (e, /, f))$$

$$L[3, 3] = ((c, /, d), +, (e, /, f)))$$

Figure 9.3: Lista aritmetičkog izraza $a - b(c/d + e/f)$ i neke njene podliste

Linearna lista se može predstaviti nizom sa slobodnim pristupom elementima ili dinamičkom strukturom sa sekvencijalnim pristupom elementima. U drugom slučaju je reč o *povezanoj listi*.

Jednostruko povezana lista je lista u kojoj svaki element liste sadrži pokazivač na sledeći element liste. Ovakva implementacija liste omogućuje jednostavno unošenje i izbacivanje elementa, dok je pronalaženje elementa sa nekim zadatim svojstvom otežano. Grafički se jednostruko povezana lista može predstaviti kao na slici 9.4.

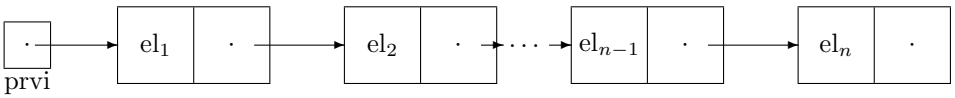


Figure 9.4: Jednostruko povezana lista

U listi se zna prvi i poslednji element, i koji element za kojim sledi. Lista može biti i prazna.

Primer. Razmotrimo konstrukciju "baze podataka" o određenoj grupi ljudi. Neka

su osobe predstavljene strukturama sa članovima ime, prezime, brlk, itd. Može se formirati lanac ili **povezana lista** takvih struktura dodavanjem člana pokazivačkog tipa:

```
struct osoba {
    char ime[20];
    char prezime[20];
    ⋮
    struct osoba *sled;
    ⋮
}
struct osoba *prvi;
```

Grafički, povezana lista od n osoba ima oblik (sl. 9.5) ili (9.6):

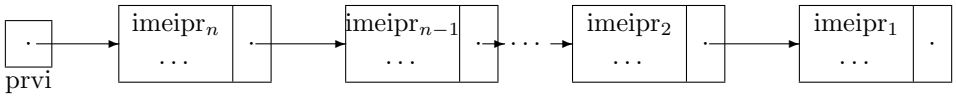


Figure 9.5: Lista osoba – redosled obrnut od redosleda unošenja

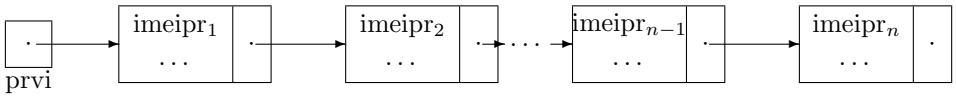


Figure 9.6: Lista osoba – redosled jednak redosledu unošenja

Promenljive tipa osoba nemaju eskplicitno ime, već im se obraćanje vrši preko pokazivača koji pokazuje na njih.

Promenljiva "prvi" pokazuje na prvi element liste kome je moguć pristup; to može biti ili prvi ili poslednji upisani element, zavisno od načina upisa novog elementa. Vrednost člana sled poslednje osobe je NULL.

Primer. Pretpostavimo da struktura tipa osoba ima i polje brlk (broj lične karte), i pretpostavimo da niz lk sadrži n brojeva ličnih karata (za n osoba). Tada se gornji lanac može konstruisati sledećim programskim segmentom:

```
struct osoba {
    char ime[20];
    char prezime[20];
    char brlk[13];
    ⋮
    struct osoba *sled;
```

```
        ⋮
}
{
    struct osoba *prvi, *p;
    int i;
    prvi=NULL;
    for(i=0; i<n; i++) {
        p=(struct osoba *)malloc(sizeof(*p));
        strcpy(p–>brlk, lk[i]);
        p–>sled=prvi;
        prvi=p;
    }
}
```

Ovako kreirana lista, na koju pokazuje pokazivač prvi, ima svojstvo da se svaki novi element dodaje na početak liste (obrnut redosled u odnosu na redosled u nizu lk).

Ako želimo da elementi liste budu poređani kao u nizu (ili datoteci) iz kojeg se čitaju podaci, potrebno je da pokazivač p pokazuje na poslednji element liste iza kojeg se dodaje sledeći element, i potreban je još jedan pokazivač – np ("novi poslednji") pomoću kojega će biti kreiran novi element. Dakle, p i np biće pokazivači na susedne elemente liste. Ovakva struktura podataka može se izgraditi na sledeći način:

```
{
    ...
    prvi=(...*)malloc(sizeof(*prvi));
    strcpy(prvi–>brlk, lk[0]);
    prvi–>sled=NULL;
    p=prvi;
    for(i=1; i<=n–1; i++) {
        np=(...*)malloc(sizeof(*np));
        strcpy(np–>brlk, lk[i]);
        np–>sled=p–>sled;
        p–>sled=np;
        p=np;
    }
}
```

U vezi sa kreiranjem, održavanjem i pretraživanjem lista postavljaju se sledeća pitanja:

1. Kako se dodaje element listi (npr. nova osoba), tj.

    a) kako se alocira prostor za novi element i

b) kako se premeštanjem pokazivača novi element ubacuje na određeno mesto u listi?

2. Kako se iz liste izbacuje (briše) element koji sledi za elementom na koji pokazuje neki pokazivač p?

3. Kako se pristupa pojedinom elementu u listi – npr. osobi sa određenim svojstvom, radi čitanja podataka o njoj, upisa novog elementa iza nađenog ili brisanja elementa koji sledi za nađenim?

Odgovori:

1. a) Alokacija prostora za novi element vrši se pozivom neke od funkcija malloc, calloc, realloc, npr. np=(...∗) malloc(sizeof(∗np)).

   b) Prethodno navedenim iskazima dodele (np–>sled=p–>sled; p–>sled=np;) ubacuje se nova struktura na koju pokazuje pokazivač np, posle strukture na koju pokazuje p (u našem slučaju – iza poslednje strukture, ali ako p ne pokazuje na poslednju, već na proizvoljnu strukturu do koje se došlo, struktura ∗np unosi se iza strukture ∗p). Ako su pre izvršenja navedenih iskaza dodele lista i položaj pokazivača bili kao na slici 9.7, onda je posle izvršenja tih iskaza lista oblika kao na slici 9.8.

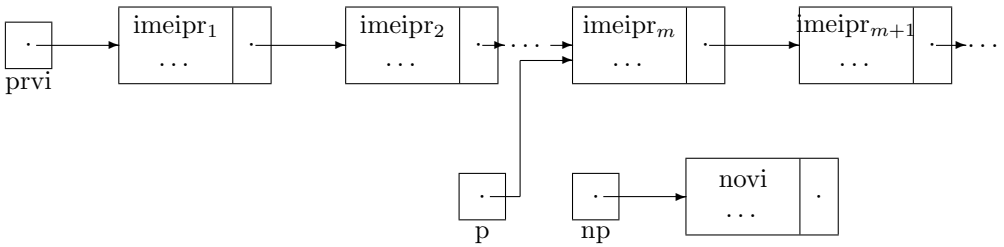


Figure 9.7: Lista pre umetanja novog elementa

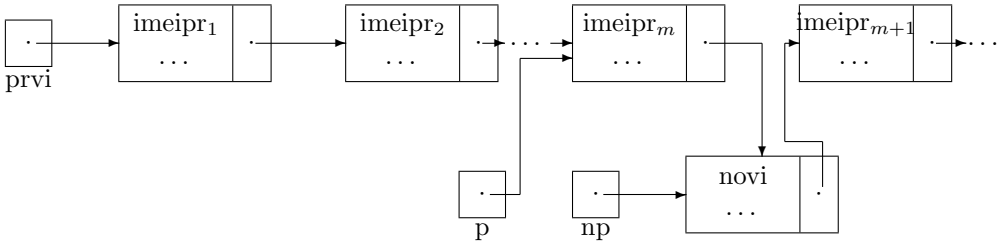


Figure 9.8: Lista posle umetanja novog elementa

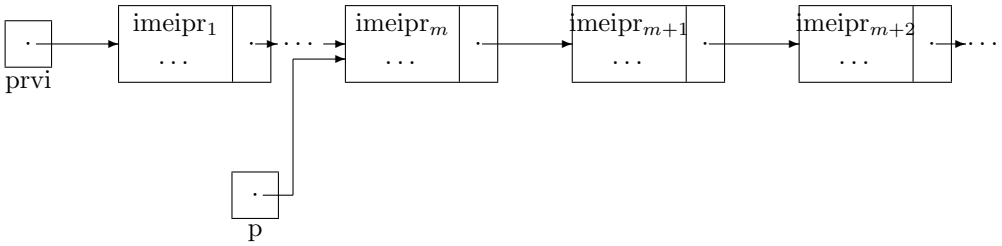


Figure 9.9: Lista pre izbacivanja elementa

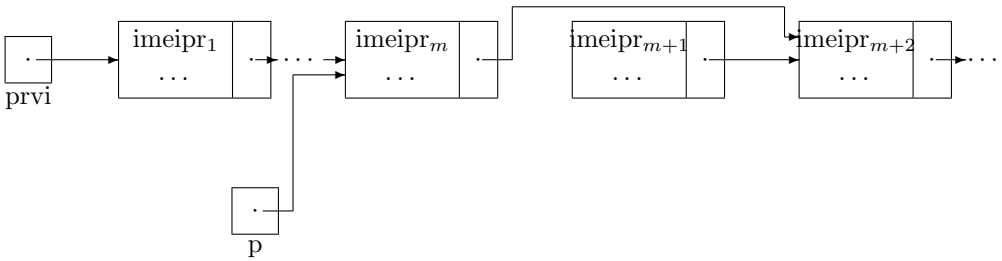


Figure 9.10: Lista posle izbacivanja elementa

2. Da bi se ostvarilo izbacivanje (brisnje) iz liste elementa koji sledi za elementom na koji pokazuje pokazivač p, potrebno je ostvariti transformaciju liste sa slike 9.9 u listu sa slike 9.10.

   Ova transformacija izvodi se izvršavanjem iskaza p–>sled=(p–>sled)–>sled.

   Često se lista obrađuje korišćenjem dva pokazivača (npr. p, p1) koji slede jedan za drugim (slika 9.11). Tada se briše element na koji pokazuje drugi od tih pokazivača (npr. p1), pri čemu prvi (p) pokazuje na element koji prethodi onom koji treba da se izbriše. Uklanjanje elementa iz liste sada ostvaruje iskaz dodele

   p–>sled=p1–>sled;

   Oslobađanje memorijskog prostora uklonjenog elementa vrši se funkcijom

   free(p1) (slika 9.12).

3. Neka na prvi element liste osobâ, koju želimo da pretražujemo, pokazuje pokazivač "prvi". Neka je zadatak – naći osobu čiji je broj lične karte jednak datom nizu cifara b, ako takva osoba postoji u listi.

   Jedna varijanta rešenja ovog zadatka je "zdravorazumska":

   ...

   p=prvi;

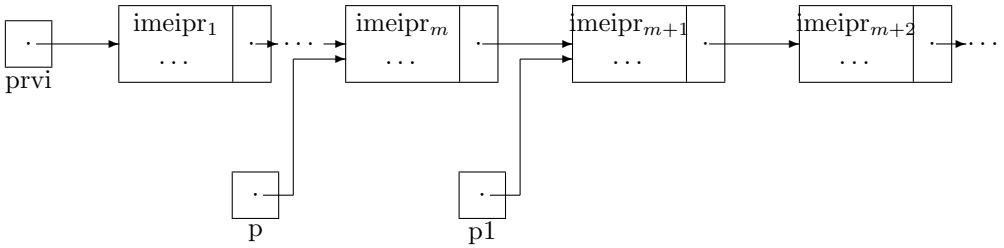


Figure 9.11: Lista sa dva susedna pokazivača

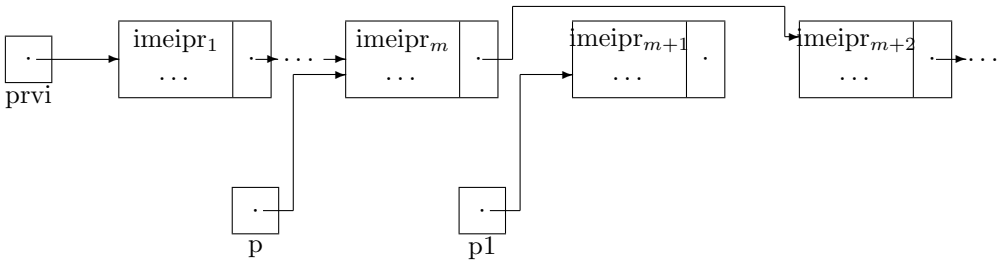


Figure 9.12: Brisanje iz liste sa dva susedna pokazivača

```
while((p! =NULL) && (strcmp(p–>brlk, b))) p=p–>sled;
```

Druga varijanta rešenja uvodi logičku promenljivu nn ("nije nađen"):

```
p=prvi; nn=1;
while (p! = NULL && nn)
    if(strcmp(p–>brlk, b) == 0) nn=0;
    else p=p–>sled;
```

Zaključak koji se može izvesti o prednostima, nedostacima i korišćenju struktura podataka koje se kreiraju, održavaju i pretražuju pomoću pokazivača: ove strukture su pogodne kada su česta ubacivanja i izbacivanja elemenata, jer nije potrebno prestrukturiranje cele liste kao što je to potrebno, na primer, u slučaju niza ili datoteke (kompletno prepisivanje). Ipak, pretraživanje ovih struktura može biti neefikasno u odnosu na uređeni niz ili na pretraživanje niza na bazi izračunljivog indeksa.

Specijalni slučajevi linearnih lista, koji se najčešće koriste u programiranju i implementaciji sistemskih programa, jesu *stek* (stog, potisna lista) i *red*. To su po prirodi dinamičke strukture (sa intenzivnim unošenjem i izbacivanjem elemenata), i mada se, kao linearne liste uopšte, mogu predstaviti nizom, efikasnija implementacija je jednostruko povezanim listama.

9.2.1 Stek

Stek je linearna lista iz koje prvi "izlazi" (uklanja se) element koji je poslednji "ušao" (koji je poslednji u listu upisan). Novi element se upisuje na "vrh" steka tj. postaje glava liste. Stek je tzv. LIFO struktura (Last In First Out – poslednji ulazi prvi izlazi).

Sa stekom su vezane tri operacije:

1. unošenje (engl. push): formira se novi čvor sa vrednošću a i u taj čvor se postavlja pokazivač na prethodnu glavu steka. Uneti element postaje nova glava steka (slika 9.13).

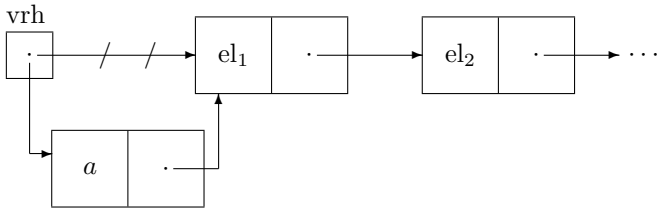


Figure 9.13: Unošenje novog elementa u stek

2. izbacivanje elementa (engl. pop): glava liste koja predstavlja stek se uklanja, a nova glava liste postaje sledeći element. Grafički (slika 9.14):

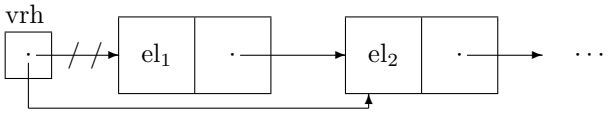


Figure 9.14: Uklanjanje elementa iz steka

3. čitanje vrednosti iz vršnog elementa steka: b= vrh–>sadrzaj.

U C-u se stek može realizovati sledećim definicijama struktura:

```
struct elsteka {
    int sadrzaj;
    struct elsteka *veza;
} *vrh;
```

Razmotrimo jedan primer korišćenja strukture steka.

Napisati funkcije za upis elementa u stek i ispis elementa iz steka, i glavnu funkciju koja, korišćenjem ovih funkcija, učitava sa standardnog ulaza cele brojeve (do pojave 0) i ispisuje ih u obrnutom poretku na standardni izlaz.

```
/*  uistek: upis u stek i ispis iz steka   */
#include <stdio.h>
```

```
#include <stdlib.h>
struct elsteka {
     int sadrzaj;
     struct elsteka *veza;
} *vrh;
void upis(int);
int ispis(void);
int x;
main()
{
     vrh=NULL;
     do {
          scanf("%d", &x);
          upis(x);
     } while(x);
     do {
          printf("%d\n", ispis());
     } while(vrh! = NULL);
}
void upis(int x)
{
     struct elsteka *p;
     p=(struct elsteka *)malloc(sizeof(*p));
     p->sadrzaj=x;
     p->veza=vrh;
     vrh=p;
}
int ispis(void)
{
     struct elsteka *p;
     x=vrh->sadrzaj;
     p=vrh;
     vrh=vrh->veza;
     free(p);
     return(x);
}
```

9.2.2 Red

Red je linearna lista iz koje prvi "izlazi" (uklanja se) element koji je prvi i "ušao" (koji je prvi u listu upisan). Novi element se upisuje na kraj reda. Zato je, pored pokazivača na početak reda ("vrh") potreban i pokazivač na kraj reda ("kraj"). Red je tzv. FIFO struktura (First In First Out – prvi ulazi prvi izlazi).

Sa redom su takođe vezane tri operacije:

1. stajanje u red – unošenje novog elementa: formira se novi čvor sa vrednošću a, a pokazivač prethodno poslednjeg elementa u redu, kao i pokazivač na kraj reda, postavljaju se na novo uneti element (slika 9.15).

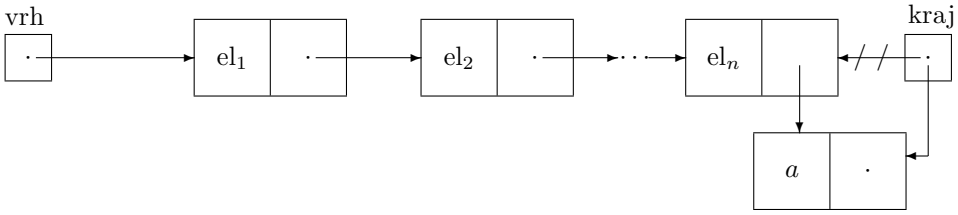


Figure 9.15: Unošenje novog elementa u red

2. izlazak iz reda – izbacivanje elementa iz reda: glava liste koja predstavlja red se uklanja, a nova glava liste postaje sledeći element. Grafički (slika 9.16):

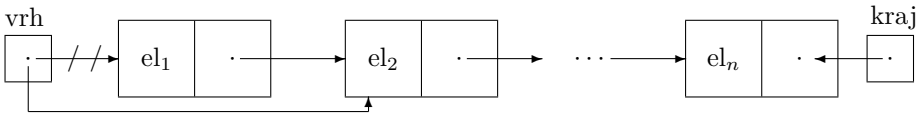


Figure 9.16: Uklanjanje elementa iz reda

3. čitanje vrednosti sa početka reda: b=vrh–>vrednost.

U C-u se struktura reda može realizovati sledećim definicijama struktura:

```
struct elreda {
    int sadrzaj;
    struct elreda *veza;
} *vrh, *kraj;
```

Razmotrimo primer korišćenja strukture reda.

Napisati funkcije za upis elementa u red i ispis elementa iz reda, i glavnu funkciju koja, korišćenjem ovih funkcija, učitava cele brojeve sa standardnog ulaza (bar 1, do pojave 0) i ispisuje ih u istom poretku na standardni izlaz.

```
/*  uired: upis u red i ispis iz reda   */
#include <stdio.h>
#include <stdlib.h>
struct elreda {
    int sadrzaj;
    struct elreda *veza;
} *vrh, *kraj;
void upis(int);
int ispis(void);
int x;
```

```
main()
{
    vrh=NULL;
    kraj=NULL;
    do {
        scanf("%d", &x);
        upis(x);
    } while(x);
    do {
        printf("%d\n", ispis());
    } while(vrh! = kraj);
}
void upis(int x)
{
    struct elreda *p;
    p=(struct elreda *)malloc(sizeof(*p));
    p->sadrzaj=x;
    p->veza=NULL;
    if(vrh! =NULL) kraj->veza=p;
    else vrh=p;
    kraj=p;
}
int ispis(void)
{
    struct elreda *p;
    x=vrh->sadrzaj;
    p=vrh;
    vrh=vrh->veza;
    free(p);
    return(x);
}
```

9.2.3 Dvostruko povezane liste

Da bi se postigla veća fleksibilnost u radu sa linearnim listama, moguće je u svaki čvor uključiti dva pokazivača – lveza i dveza, koji pokazuju na čvorove s obe strane tog čvora. Dve pokazivačke promenljive, levi i desni, pokazuju na prvi odnosno poslednji element liste (tj. levi i desni kraj liste). Tako se dobija dvostruko povezana lista (slika 9.17).

Pošto prvi čvor u listi nema prethodnika (tj. levog suseda), vrednost pokazivačkog člana lveza u strukturi kojom se taj čvor predstavlja jeste NULL. Slično važi i za pokazivačko polje dveza u strukturi kojom se predstavlja poslednji čvor u listi. Zbog toga se dvostruko povezana lista može modifikovati tako da

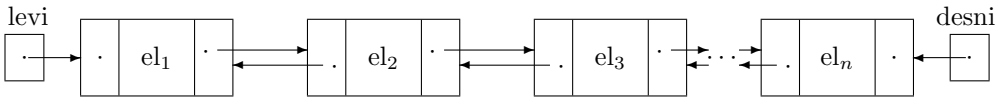


Figure 9.17: Dvostruko povezana lista

se eliminišu pokazivači na njen levi i desni kraj, a da njihovu ulogu preuzmu ovi NULL pokazivači. Tako se dobija ciklična lista prikazana na slici 9.18.

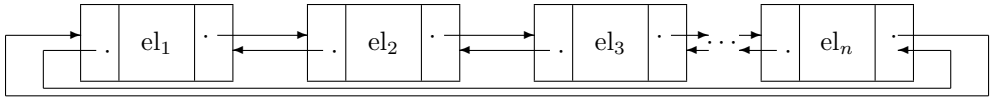


Figure 9.18: Ciklična dvostruko povezana lista

Uobičajeno je povezanu listu "dopuniti" jednim "praznim" čvorom koji je uvek prisutan u listi čak kada se iz nje uklone svi njeni pravi elementi. Ovaj fiktivni čvor omogućuje uniformnu obradu liste bez obzira da li je ona prazna ili ne.

Za cikličnu dvostruko povezanu listu važi da je dveza(lveza(x)) = lveza(dveza(x)), za svaki čvor x.

U C-u se dvostruko povezana lista može realizovati sledećim tipom strukture (i promenljivih):

```
struct element {
    int sadrzaj;
    struct element *lveza, *dveza;
} *levi, *desni, *p, *x;
```

Novi čvor se unosi, sleva ili zdesna od čvora na koji imamo pokazivač x, sledećim iskazima:

sleva:

(1) p=(struct element $*$)malloc(sizeof($*p$));
(2) p–>lveza = x–>lveza;
(3) p–>dveza = x;
(4) (x–>lveza)–>dveza = p;
(5) x–>lveza = p;

zdesna:

(1) p=(struct element $*$)malloc(sizeof($*p$));
(2) p–>lveza = x;
(3) p–>dveza = x–>dveza;
(4) (x–>dveza)–>lveza = p;
(5) x–>dveza = p;

Redosled poslednja dva iskaza dodele je bitan, jer bi se u obrnutom redosledu prvo (iskaz (5)) promenio sadržaj člana x–>lveza (u slučaju dodavanja sleva) –

pokazivač na levog suseda čvora x, pa onda ne bi moglo da se dođe do pokazivačkog polja dveza na desnog suseda tog levog suseda (iskaz (4)). Slično važi i kod unošenja zdesna.

Za uklanjanje nekog čvora iz dvostruko povezane liste dovoljno je imati pokazivač x na taj čvor:

```
(x–>lveza)–>dveza = x–>dveza;
(x–>dveza)–>lveza = x–>lveza;
free(x);
```

(Kod jednostruko povezane liste trebalo je imati pokazivač na čvor – prethodnik).

9.3 Dinamičke strukture podataka opšteg tipa

Strukture podataka koje se mogu predstaviti opštom strukturom grafa nazivaćemo *grafoidnim strukturama podataka.*

Uvedimo prvo neke osnovne pojmove teorije grafova.

Graf $\Gamma = (A, R)$ je uređeni par čija je prva komponenta A skup a druga komponenta R je binarna relacija nad skupom A, $R \subseteq A \times A$. Za skup A podrazumevaćemo da je konačan. Elemente skupa A nazivamo *čvorovima* (ili temenima) a elemente skupa R – *granama* (ili potezima) grafa Γ. Ako je uređeni par $(a, b) \in R$, kaže se da je čvor a izlazni čvor grane (a, b), a čvor b – ulazni čvor grane (a, b). *Izlazni red* čvora a je broj grana kojima je a izlazni čvor. *Ulazni red* čvora a je broj grana kojima je čvor a ulazni čvor.

Graf može biti *neusmeren* ili *usmeren.* Graf je neusmeren ako je relacija R simetrična, tj. ako $(a, b) \in R$ povlači $(b, a) \in R$, za svaki par čvorova $a, b \in A$. U suprotnom, graf je usmeren.

Prirodan način za predstavljanje grafa je dijagramski, pri čemu se čvorovi grafa predstavljaju kružićima u koje su upisani elementi skupa A – "imena" čvorova, a grane grafa se predstavljaju dužima koje spajaju parove kružića – čvorova. Ako je graf usmeren, duži kojima se predstavljaju grane grafa takođe su usmerene, tj. imaju strelicu na kraju koji odgovara ulaznom čvoru te grane. Ako je graf neusmeren, takve su i duži kojima se predstavljaju grane grafa.

Na primer, ako je $A = \{a, b, c, d\}$, $R = \{(a, b), (a, c), (b, a), (b, d), (c, b), (c, d), (d, a), (d, c)\}$, graf $\Gamma = (A, R)$ može se dijagramski predstaviti kao na slici 9.19:

Putanja dužine $k - 1$ $(k > 1)$ iz čvora a_1 u čvor a_k u grafu Γ jeste niz čvorova $a_1, a_2, \ldots, a_k$ (moguće je da se čvorovi ponavljaju) povezanih granama $(a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, a_k)$ (grane se obično takođe smatraju delom putanje). *Putanja dužine 0* je svaki pojedinačni čvor a_i. Čvor u je *dostižan* iz čvora v ako postoji putanja iz v u u dužine ≥ 0.

Svaki usmereni graf $\Gamma = (A, R)$ može se "dopuniti" do neusmerenog grafa, dodavanjem najmanjeg broja grana koje dovode do simetričnosti relacije R. Graf je *povezan* ako, dopunjen do neusmerenog grafa, ima sledeće svojstvo: iz svakog čvora vodi putanja dužine ≥ 0 do svakog drugog čvora.

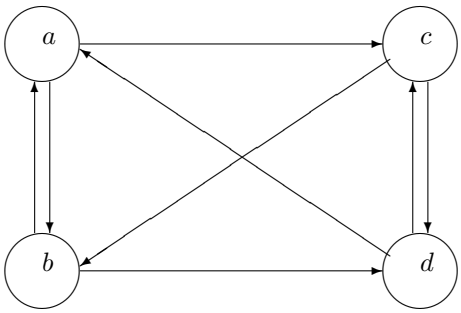


Figure 9.19: Primer grafa

Ciklus je putanja dužine > 1 čiji se prvi i poslednji čvor poklapaju.

Dijagramski način predstavljanja grafa nije pogodan za računar, pa se primenjuju razni drugi načini predstavljanja.

Statičke stukture podataka Jedan način za predstavljanje grafa je matrični. Graf $\Gamma = (A, R)$ čiji skup A sadrži n elemenata predstavlja se matricom $n \times n$, u kojoj se u preseku i-te vrste i j-te kolone nalazi 1 ako je $(a_i, a_j) \in R$, inače 0. Ova matrica zahteva prostor veličine n^2 bez obzira na broj grana u grafu. U opštem slučaju dosta je retka (ima dosta 0-a), tj. neefikasno koristi prostor. Prethodni primer grafa bi se matrično predstavio matricom na slici 9.20:

	a	b	c	d
a	0	1	1	0
b	1	0	0	1
c	0	1	0	1
d	1	0	1	0

Figure 9.20: Matrični prikaz grafa

Algoritam za matrično predstavljanje grafa $\Gamma = (A, R)$ zadatog skupovima A i R ($|A| = n$) može imati sledeći oblik:

1. numerisati elemente skupa A sa $a_1, a_2, \ldots, a_n$; preći na sledeći korak;

2. svim elementima matrice M sa n vrsta i n kolona dodeliti 0 ($M_{i,j} \leftarrow 0$, za $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots n$); preći na sledeći korak;
3. za svaku vrednost $i = 1, 2, \ldots, n$ uraditi sledeće:
   3.a. za svaku vrednost $j = 1, 2, \ldots, n$ uraditi sledeće:
      3.a.a ako je $(a_i, a_j) \in R$ onda $M_{i,j} \leftarrow 1$;
4. kraj.

Umesto eksplicitnog predstavljanja svih 0-a u matrici, moguće je predstaviti samo 1-ce nekom strukturom podataka koja omogućuje jednom elementu da ukaže na sledeći. Najjednostavnija struktura podataka sa ovim svojstvom je povezana lista (o povezanim listama v. opširnije u odeljku 9.2, "Povezane liste"). U ovoj listi, svaki čvor je pridružen listi koja se sastoji od svih grana koje izlaze iz tog čvora. Tako se čitav graf predstavlja nizom lista. Svaki element niza sastoji se od oznake čvora i pokazivača na početak liste grana koje iz njega izlaze. U slučaju statičnog grafa – bez unošenja ili izbacivanja, liste se mogu predstaviti nizovima. Čvorovi se, pre svega, numerišu brojevima od 1 do n. Za predstavljanje grafa Γ može se koristiti niz od $|A| + |R|$ elemenata. Prvih $|A|$ elemenata odgovara čvorovima u datom poretku (prvi, drugi, itd.), a svaki element sadrži indeks elementa niza gde počinje lista grana koje izlaze iz tog čvora. U prethodnom primeru, ako se čvorovi a–d "prenumerišu" indeksima 1–4, onda u ovoj reprezentaciji graf ima oblik kao na slici 9.21:

5	7	9	11	2	3	1	4	2	4	1	3
1	2	3	4	5	6	7	8	9	10	11	12

Figure 9.21: Alternativna reprezentacija grafa

Prostor koji zauzima graf predstavljen pomoću niza lista je manji nego korišćenjem matrice, ali su programi za rad sa ovom reprezentacijom složeniji.

Dinamičke strukture podataka Uopštena grafoidna struktura sa n čvorova može se implementirati tipom strukture sa jednim članom koji odgovara čvoru grafa i n članova pokazivačkog tipa koji odgovaraju izlaznim granama iz tog čvora. U C-u se ta definicija može zadati na sledeći način:

```
sruct cvor {
    char znak;
    struct cvor *poteg[BRCVOR];
};
```

Sam graf se sada može deklarisati kao niz čiji su elementi tipa cvor, ili dinamičkim promenljivim tipa cvor:

```
struct cvor graf[BRCVOR];
```

Ova definicija tipa cvor i promenljive graf, za primer grafa $\Gamma = (A, R)$, gde je $A = \{a, b, c, d\}$ a $R = \{(a, b), (b, c), (b, d), (c, a), (c, d), (d, c)\}$, $(n = 4, m = 6)$ može se grafički predstaviti kao na slici 9.22.

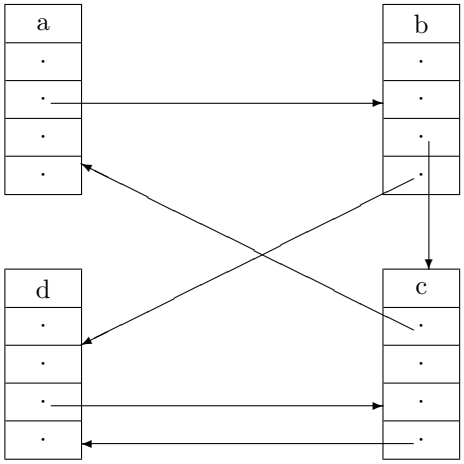


Figure 9.22: Grafički prikaz prve implementacije grafa

U ovakvoj implementaciji grafa, najčešće će veliki broj polja pokazivačkog tipa biti NULL, (kadgod jedan čvor nema granu prema drugom čvoru). Zato se uopštena grafoidna struktura može definisati i na drugi način: čvorovima (numerisanim od 1 do n), i listama grana (po jedna lista za svaki čvor) kojima je taj čvor izlazni. Čvorovi se predstavljaju pokazivačkim tipom (na odgovarajuće liste grana), a grane se predstavljaju tipom strukture sa dva člana: jedan član sadrži oznaku ulaznog čvora grane, a drugi – pokazivač na sledeću granu u listi. Ako pored broja čvorova, n, uzmemo u obzir i broj grana, m, u C-u se ova definicija grafa može zadati na sledeći način:

```
#define BRCVOR .../*    npr. 10    */
#define BRGRANA .../*    npr. 20    */
struct grana {
    char ulcvor;
    struct grana *sledgrana;
}
```

Sam graf se sada može definisati kao par nizova čiji su elementi čvorovi odnosno grane grafa:

```
struct grana *cvorovi[BRCVOR];
struct grana grane[BRGRANA];
```

Prethodni primer grafa, zadat ovim definicijama tipa (*grana*) i promenljivih (nizovi *cvorovi* i *grane*), može se grafički predstaviti kao na slici 9.23.

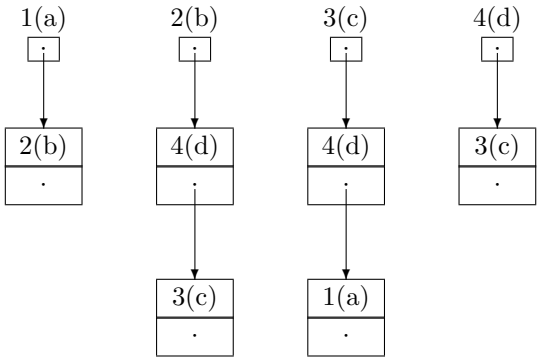


Figure 9.23: Grafički prikaz druge implementacije grafa

9.3.1 Obilazak grafa

Za razliku od sekvencijalnih struktura, kakav je niz ili lista, čijim elementima se pristupa redom, redosled kojim se pristupa elementima grafa – čvorovima odnosno granama – predstavlja problem. U raznovrsnim zadacima nad grafovima potrebno je, u cilju specifične obrade definisane specifičnim zadatkom, "obići graf", tj. pristupiti svim čvorovima i granama grafa koristeći samo postojeće putanje u grafu. Dve najčešće korišćene tehnike obilaska grafa jesu obilazak grafa po dubini (engl. depth first search, DFS), i obilazak grafa po širini (engl. breadth first search, BFS).

Obilazak grafa po dubini – DFS Ova tehnika obilaska neusmerenog grafa sastoji se u sledećem: svi čvorovi grafa $\Gamma = (A, R)$ na početku se smatraju neobeleženim – neposećenim; polazi se od nekog – proizvoljnog čvora u grafu, $v \in A$, on se označava obeleženim, a zatim se, za svaku granu koja povezuje čvor v sa nekim čvorom w, taj čvor w, ako je neobeležen, posmatra kao polazni i na njega primenjuje (rekurzivno) ista tehnika. Kada se tehnika primeni na sve grane koje prolaze kroz čvor v, taj čvor se napušta i nivo rekurzije smanjuje. Tehnika se naziva obilazak grafa po dubini jer se sa obilaskom ide unapred (u dubinu grafa) sve dok je to moguće. Ako je graf neusmeren i povezan, ovim postupkom obilaze se svi čvorovi i sve grane grafa. Za usmereni graf postupak je isti, ali se može dogoditi da se obilaskom ne dođe do svakog čvora ili svake grane.

Na primer, tehnikom obilaska grafa po dubini, u grafu sa slike 9.24 čvorovi bi bili posećeni u redosledu a, b, e, b, f, j, g, j, f, b, a, c, h, i, d, i, h, c, a. Ako kao cilj obilaska grafa postavimo numeraciju čvorova i to *ulaznu numeraciju* – ako beležimo redosled u kome smo prvi put pristupili pojedinim čvorovima (u kome smo "ušli" u čvorove), ili *izlaznu numeraciju* – ako beležimo redosled u kome smo poslednji put pristupili čvorovima (u kome smo "napustili" čvorove), onda će u ulaznoj numeraciji čvoru a biti dodeljen broj 1 (od njega počinje obilazak grafa), čvoru b – broj 2, čvoru e – broj 3, itd. (v. sliku 9.24); izlaznu numeraciju možemo da dobijemo čitajući navedeni redosled zdesna ulevo i dodeljujući poslednjim pojavljivanjima

čvorova opadajući niz brojeva – čvoru a broj – 10 (to je čvor u kome se završava obilazak grafa), čvoru c – 9, h – 8, i – 7, d – 6, itd.

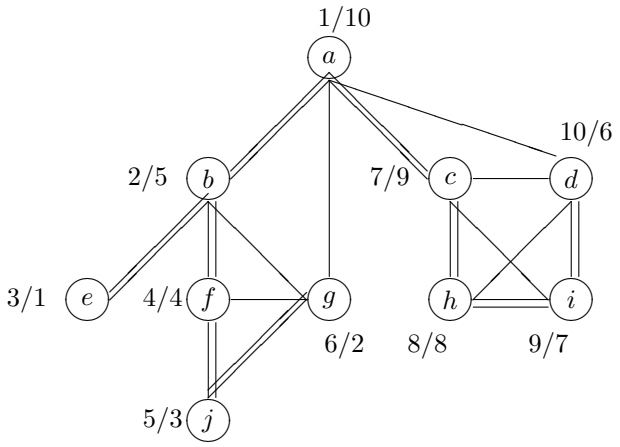


Figure 9.24: Primer obilaska grafa po dubini sa različitim ciljem (obradom): a)ulazne numeracije čvorova (prvi broj uz čvor); b)izlazne numeracije čvorova (drugi broj uz čvor); c) izdvajanja stabla grafa (podebljane grane)

Cilj obilaska grafa može da bude i neka druga ulazna odnosno izlazna obrada (pri prvom ulasku u čvor odnosno pri njegovom napuštanju). Na primer, izdvajanje stabla iz grafa (novog grafa koji sadrži sve čvorove polaznog grafa i samo grane kojima se došlo prvi put do nekog čvora – do neobeleženog čvora), pronalaženje ciklusa u grafu, itd.

Rekurzivni algoritam obilaska grafa po dubini ima sledeći oblik:

```
DFS(G, v)
{
    obeležiti v;
    izvršiti ulaznu obradu (npr. numerisati čvor v);
    for (sve grane (v, w)) {
        if (w neobeležen) {
            DFS(G, w);
            izvršiti izlaznu obradu (npr. izlaznu numeraciju, dodavanje grane
(v, w) stablu grafa, itd.)
        }
    }
}
```

Nerekurzivna varijanta algoritma DFS koristi eksplicitnu strukturu steka i pamćenje podataka na steku, koje se u rekurzivnoj varijanti organizuje sistemski.

Pretpostavimo da je (neusmeren) graf zadat listama grana (po jedna lista za svaki čvor grafa) koje direktno povezuju posmatrani čvor sa drugim čvorovima grafa (v. glavu 12). Svaka grana u listi grana koja odgovara jednom čvoru, može da bude predstavljena strukturom sa jednim članom – Cvor – oznakom čvora na drugom kraju grane, i jednim pokazivačkim članom – Sled – na sledeću granu u listi. I sam čvor (npr. v) je predstavljen strukturom koja, osim oznake čvora ima i pokazivački član na prvi element u listi pridruženih grana, $v.Prvi$. Na steku se pamte svi čvorovi na putu od korena (polaznog čvora) do tekućeg čvora; ako se na vrhu steka nalazi čvor v, pre nego što se krene u dubinu grafa po grani (v, w), tj. pre nego što se na stek stavi čvor w koji se nalazi na drugom kraju grane (v, w), na stek se stavlja pokazivač na sledeću granu iz čvora v kojom će se krenuti pošto se obilazak vrati iz čvora w (eksplicitno pamćenje adrese povratka koja se i u rekurzivnoj varijanti pamti na steku – ali sistemski).

Nerekurzivna varijanta algoritma obilaska grafa po dubini ima sledeći oblik:

```
DFS (G, v)
{
    while(postoji neobeležen čvor w) {
        if(v je neobeležen) w = v;
        obeleži w;
        izvrši ulaznu obradu;
        Grana = w.Prvi;
        staviti na stek čvor w i granu Grana;
        Prethodnik = w;
        while(stek nije prazan) {
            skini pokazivač Grana sa vrha steka;
            while(Grana! = nil) {
                Sledbenik = Grana− > Cvor;
                if (Sledbenik nije obeležen) {
```

```
                        obeleži Sledbenik;
                        izvrši ulaznu obradu na Sledbenik;
                        upiši Grana− > Sled na vrh steka;
                        Grana = Sledbenik.Prvi;
                        Prethodnik = Sledbenik;
                        upiši Prethodnik na vrh steka;
                    }
                    else {
                        izvrši izlaznu obradu za granu (Prethodnik, Sledbenik);
                        Grana = Grana− > Sled;
                    }
                }
                skini Sledbenik sa vrha steka; /∗ korak nazad ∗/
                if (stek nije prazan) {
                    neka su Grana i Prethodnik na vrhu steka;
                    izvršiti izlaznu obradu za granu (Prethodnik, Sledbenik);
                }
        }
    }
```

Obilazak grafa po širini – BFS Druga tehnika obilaska grafa je obilazak po nivoima tj. po širini. Naime, ako se sa obilaskom krene iz čvora v kao "korena", onda se obilaze svi čvorovi na drugom kraju grana iz v (čvorovi na nivou 1 od korena) pre nego što se krene na obilazak čvorova na nivou 2 od korena. Obilazak svakog čvora uključuje pamćenje grana koje kroz njega prolaze – i koje će biti obrađene u sledećim koracima, pa nema potrebe za vraćanjem u taj čvor. Zato se umesto steka u DFS algoritmu, u BFS algoritmu koristi red tj. FIFO struktura – čvorovi koji prvi uđu prvi i izlaze sa te strukture. Algoritam je sličan nerekurzivnoj varijanti DFS-algoritma i ima sledeći oblik:

```
BFS(G, v)
{
    obeleži v;
    upiši v u red;
    while(red nije prazan) {
        skini prvi čvor w iz reda;
        izvrši ulaznu obradu na w;
        for (sve grane (w, x) za koje x nije obeležen {
            obeleži x;
            izvršiti izlaznu obradu grane (w, x) – na primer, dodati tu granu
stablu grafa;
            upisati x u red;
        }
    }
}
```

9.3.2 Drvoidne strukture

Specijalni oblik grafa je drvo.

Drvo je usmeren graf u kome:

1) postoji jedinstven čvor – *koren drveta*, u koji ne ulazi nijedna grana;

2) u sve ostale čvorove ulazi tačno po jedna grana a izlazi proizvoljno mnogo;

3) ne sadrži cikluse.

Čvorovi iz kojih izlazi 0 grana nazivaju se *listovima* drveta.

Drvo se može definisati i rekurzivno sledećom definicijom:

1) Skup $\{n\}$ koji se sastoji od jednog čvora je drvo;

2) ako je n – čvor a $T_1, T_2, \ldots, T_k$ – drveta, onda je drvo i graf koji se dobije dodavanjem grana iz čvora n kao korena do korena drveta $T_1, T_2, \ldots, T_k$.

$T_1, T_2, \ldots, T_k$ su sada *poddrveta* novog drveta.

Rekurzivna definicija drveta se koristi u rekurzivnim algoritmima obrade drveta.

Jedan specijalni oblik drveta je *linearno drvo*, kod koga u svaki čvor ulazi i iz svakog čvora izlazi najviše po jedna grana.

Drugi specijalni oblik drveta je *binarno drvo*, koje se najčešće koristi u primenama. Binarno drvo je najpogodnije definisati rekurzivno i obrađivati (kreirati, modifikovati, pretraživati) rekurzivnim algoritmima.

Definicija: Binarno drvo je konačni skup čvorova koji je:

1) prazan ili

2) sastoji se od korena sa dva disjunktna binarna poddrveta – *levog* i *desnog* poddrveta.

Algoritmi za kreiranje i obilazak (pretraživanje) binarnog drveta mogu biti trojaki ("kreirati" ili "obići" u ovim algoritmima će se zameniti jedinstvenim terminom "obraditi"):

1. *sa_vrha* (engl. pre-order):

   1) ako je drvo prazno – završiti;

   2) inače, obraditi koren, zatim obraditi levo poddrvo, a zatim desno poddrvo, i završiti.

2. *sleva_nadesno* (engl. in-order):

   1) ako je drvo prazno – završiti;

   2) inače, obraditi levo poddrvo, zatim obraditi koren, a zatim obraditi desno poddrvo i završiti.

3. *sa_dna* (engl. post-order):

   1) ako je drvo prazno – završiti;

   2) inače, obraditi levo poddrvo, zatim obraditi desno poddrvo, a zatim obraditi koren i završiti.

Strukture podataka koje se predstavljaju drvetom nazivaju se *drvoidnim strukturama podataka*. Njih je najefikasnije implementirati dinamičkim promenljivim tipa strukture sa tri člana: info (informacija, npr. slovo), pokazivač na koren levog poddrveta, pokazivač na koren desnog poddrveta.

Primer. Napisati program koji kreira binarno drvo algoritmom sa_vrha i obilazi ga algoritmima sa_vrha, sleva_nadesno i sa_dna. Drvo je zadato nabrajanjem čvorova (na primer slova) na standardnom ulazu, počevši od korena, preko čvorova levog poddrveta, završno sa čvorovima desnog poddrveta. Na mestu praznog poddrveta nalazi se znak "·". Na primer, ako se na standardnom ulazu nađu karakteri "abc...de..f.g..", oni odgovaraju drvetu koje je grafički predstavljeno na slici 9.26.

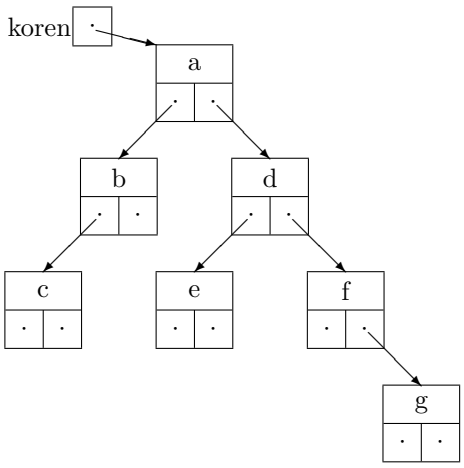


Figure 9.25: Grafički prikaz primera binarnog drveta

Obilaskom drveta algoritmima sa_vrha, sleva_nadesno i sa_dna treba na standardnom izlazu dobiti rezultat abcdefg, cbaedfg, cbegfda, redom.

```
/* obilazak: obilazak drveta */
#include <stdio.h>
#include <stdlib.h>
struct cvor {
     char info;
     struct cvor *lveza, *dveza;
} *koren;
char zn;
void svrha(struct cvor *p);
void sadna(struct cvor *p);
void slevanad(struct cvor *p);
struct cvor * unesi(void);
```

```
main()
{       koren=unesi();
        printf("\n");
        svrha(koren);
        printf("\n");
        slevanad(koren);
        printf("\n");
        sadna(koren);
        printf("\n");
}
void svrha(struct cvor *p)
{       if(p! =NULL) {
                putchar(p–>info);
                svrha(p–>lveza);
                svrha(p–>dveza);
 /*              free(p); */
        }
}
void slevanad(struct cvor *p)
{       if(p! =NULL) {
                slevanad(p–>lveza);
                putchar(p–>info);
                slevanad(p–>dveza);
 /*              free(p); */
        }
}
void sadna(struct cvor *p)
{       if(p! =NULL) {
                sadna(p–>lveza);
                sadna(p–>dveza);
                putchar(p–>info);
                free(p);
        }
}
struct cvor * unesi(void)
{       struct cvor *p;
        zn=getchar();
        if(zn! =".") {
                p=(struct cvor * )malloc(sizeof(*p));
                p–>info=zn;
                p–>lveza=unesi();  p–>dveza=unesi();
        }
        else p=NULL;
        return p;
}
```

Prethodni pirmer pokazuje da algoritam pre-order obilaska binarnog drveta (sa vrha) omogućuje jednoznačno kreiranje binarnog drveta i to tako što se niz znakova - obeležja čvorova - čita redom, jedan po jedan. Koreni čvor je obeležen prvim navedenim znakom.

Algoritam post-order obilaska binarnog drveta (sa dna) omogućuje jednoznačno kreiranje binarnog drveta iz zadatog niza znakova, ali zahteva poznavanje celog niza obeležja čvorova pre nego što se krene u kreiranje binarnog drveta sa tim čvorovima. Koreni čvor je obeležen poslednjim navedenim znakom. Tako, ako je niz "..c.b..e...gfda" – niz obeležja čvorova binarnog drveta u post-order poretku, onda se iz njega može jednoznačno rekonstruisati binarno drvo prikazano na slici 9.26.

Sledeće definicije tipa, promenljivih i funkcija to omogućuju:

```
struct cvor {
      char info;
      struct cvor *lveza, *dveza;
} *koren;
char zn[100];
int n;
struct cvor * unesi(void)
{      struct cvor *p;
       if(zn[n]! ="."") {
             p=(struct cvor *)malloc(sizeof(*p));
             p->info=zn[n--];
             p->dveza=unesi();  p->lveza=unesi();
       }
       else { n--; p=NULL; }
       return p;
}
int main()
{      int i = 0;
      while ((zn[i++]=getchar())!='\n');
      n = i-2;
      koren=unesi();
}
```

Algoritam in-order, sam za sebe, ne omogućuje jednoznačno kreiranje binarnog drveta iz zadatog niza znakova. Na primer, nizu znakova (".” opet NULL pokazivač) ".c.b.a.e.d.f.g." odgovara binarno drvo na slici **??** ali i ranije prikazano binarno drvo na slici 9.26, jer se u nizu znakova ne zna koji odgovara korenu (u prvom slučaju to je "d", u drugom "a").

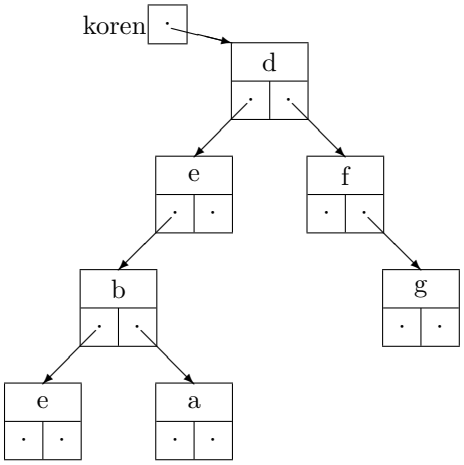


Figure 9.26: Jedno od binarnih drveta kreiranih algoritmom in-order od zadatog niza znakova

10

Pretraživanje dinamičkih struktura podataka

Kao što smo već ranije videli, pretraživanje je postupak pronalaženja elementa sa određenim svojstvom u zadatom skupu elemenata, ili ustanovljavanja da takvog elementa u zadatom skupu nema. Skup može biti zadat, kao i kod sortiranja, različitim strukturama podataka kao što su niz, lista, datoteka, drvo, itd. Naglasimo ovde razliku između ovako uvedenog pojma strukture podataka i pojma strukture (struct) u C-u koja odgovara, na primer, slogu u Pascal-u. Svaki element strukture podataka može biti slog tj. C-struktura i u tom slučaju svaka struktura ima član (ključno polje, ključ) koje se koristi u pretraživanju. Cilj pretraživanja je tada, za zadati *ključ pretraživanja* (vrednost tipa ključnog polja), naći sve strukture sa vrednošću ključa koja se poklapa sa ključem pretraživanja. Postoji mogućnost da više struktura ima istu vrednost ključa, što se može razrešiti, npr. uvezivanjem u listu takvih struktura.

Za pretraživanje se obično vezuje nekoliko zadataka i funkcija:

- inicijalizovati strukturu podataka koja se pretražuje
- tražiti strukturu (ili strukture) koja ima dati ključ
- umetnuti novu strukturu (ako nije pronađena)
- izbrisati strukturu sa datom vrednošću ključa

Pretraživanje može biti, kao i sortiranje, unutrašnje i spoljašnje. Ovde ćemo se baviti samo unutrašnjim pretraživanjem i to dinamičkih struktura – lista i drveta.

Pretraživanje se može zasnivati na poređenju ključeva, ili na cifarskim karakteristikama ključa. Poređenje ključeva, pak, može se odnositi na prave vrednosti ključa (npr. sekvencijalno, binarno pretraživanje, pretraživanje binarnog drveta), ili na transformisane vrednosti ključa (direktno, heš pretraživanje). Pretraživanje zasnovano na cifarskim karakteristikama ključa je *cifarsko pretraživanje* (radix pretraživanje, pretraživanje po osnovi), i ovde neće biti posebno razmatrano.

10.1 Sekvencijalno pretraživanje liste

Prva tri zadatka pretraživanja, u slučaju da je struktura podataka lista sortirana po ključnom polju, može se realizovati sledećim funkcijama:

```
struct cvor {
    int kljuc, info;
    struct cvor *sled;
} *glava, *t, *z;

void inicijal(void)
{
    z= (struct cvor *)malloc(sizeof(*z));
    z->sled = z;
    glava = (struct cvor *)malloc(sizeof(*glava));
    glava->sled = z;
}

struct cvor *listinsert(int k, struct cvor *t)
{
    struct cvor *x;
    z->kljuc = k;
    while((t->sled)->kljuc < k) t = t->sled;
    x=(struct cvor *)malloc(sizeof(*x));
    x->sled = t->sled;
    t->sled = x;
    x->kljuc = k;
    return x;
}

struct cvor *listpret(int k, struct cvor *t)
{
    z->kljuc = k;
    do { t = t->sled; }
    while(k>t->kljuc);
    if(k==t->kljuc) return t;
    else return z;
}
```

Ovde se pretpostavlja da je lista uređena u neopadajući poredak, da čvor (struktura) *z služi kao fiktivni član – u pretraživanju kao n+1.-vi element koji dobija vrednost traženog ključa. Pretraživanje ide samo do ključa koji je jednak ili veći od traženog ključa (k). Unošenje novog elementa vrši se ispred prvog elementa čiji je ključ >=k, a pošto je u jednostruko povezanoj listi jednostavno unošenje samo iza elementa na koji imamo pokazivač, pokazivač t se pomera na sledeći element tek ako smo utvrdili da je i sledeći element manji od k (while((t->sled)->kljuc < k)).

Pretraživanje zahteva u proseku $n/2$ koraka.

Za vežbu: napisati program koji testira prethodne funkcije.

10.2 Pretraživanje binarnih drveta

Metoda koja se, mada se klasifikuje u elementarne zbog svoje jednostavnosti, vrlo često koristi. Svaki čvor uređenog binarnog drveta ima osobinu da je vrednost njegovog ključnog polja veća od vrednosti ključnog polja svakog čvora njegovog levog poddrveta, i da je manja ili jednaka sa vrednošću ključnog polja svakog čvora njegovog desnog poddrveta.

U implementaciji ovog algoritma pretraživanja koriste se strukture sa članom koji sadrži podatak (ključno polje) i dva pokazivačka člana, l (levo) i d (desno) na korene levog odnosno desnog poddrveta. Ako struktura nema levo ili desno poddrvo, odgovarajući pokazivač treba da ima vrednost NULL. U implementaciji se, međutim, slično sekvencijalnom pretraživanju liste, uvodi dodatni (fiktivni) čvor na koji pokazuju svi pokazivači koji treba da imaju vrednost NULL. U ovaj fiktivni čvor upisuje se, pri pretraživanju, i vrednost ključa koji se traži, kako bi se pretraživanje uvek uniformno završilo – nalaženjem pokazivača na strukturu sa traženom vrednošću. Pokazivač na koren drveta neka je glava. Dakle, u implementaciji će se koristiti sledeći tipovi i promenljive:

```
struct cvor {
    int kljuc, info;
    struct cvor *l, *d;
} *z, *glava;
```

Funkcija inicijalizacije, slično inicijalizaciji liste, sadrži kreiranje fiktivnog čvora sa levim i desnim pokazivačem koji pokazuju na sâm taj čvor, kao i postavljanje pokazivača na koren drveta – glava – da pokazuje na taj fiktivni čvor (drvo je pri inicijalizaciji prazno):

```
void inicijal(void)
{
    z=(struct cvor *)malloc(sizeof(*z));
    z->l = z;
    z->d = z;
    glava = z;
}
```

Pretraživanje binarnog drveta, s obzirom na svojstvo binarnog drveta, počinje poređenjem podatka u korenom čvoru drveta sa ključem k koji se traži. Ako su vrednosti jednake, pretraživanje je uspešno. Ako je vrednost ključa manja od vrednosti podatka u čvoru drveta, pretraživanje se nastavlja u levom poddrvetu. Ako je vrednost ključa veća od vrednosti podatka u čvoru drveta, pretraživanje se nastavlja u desnom poddrvetu. Dakle, pretraživanje binarnog drveta je po suštini rekurzivni postupak, i može se realizovati sledećom rekurzivnom funkcijom:

```
struct cvor *drvopretr(int k, struct cvor *x)
{
```

```
    z–>kljuc = k;
    if(k < x–>kljuc) return drvopretr(k, x–>l);
    else if(k > x–>kljuc) return drvopretr(k, x–>d);
    else return x;
}
```

Odgovarajuća iterativna funkcija ima sledeću deklaraciju:

```
struct cvor *drvopreti(int k, struct cvor *x)
{
    z–>kljuc = k;
    while(k ! = x–>kljuc)
        if(k < x–>kljuc) x = x–>l;
        else x = x–>d;
    return x;
}
```

Sve strukture sa ključem jednakim k mogu se pronaći uzastopnim postavljanjem t na drvopretr(k,t) (odnosno drvopreti(k,t)), počevši od pokazivača glava, kao za sekvencijalno pretraživanje:

```
t=glava;
while(t! =z) {
    t=drvopret(k,t);
    t=t–>d
}
```

Slično, rekurzivna i iterativna funkcija za unošenje novog podatka u binarno drvo, s obzirom na svojstvo binarnog drveta, ima sledeći oblik:

```
struct cvor *drvoinsertr(int k, struct cvor *x)
{
    struct cvor *f;
    z–>kljuc = k;
    f=x;
    if(k<x–>kljuc) x=x–>l;
    else x=x–>d;
    if (x==z) {
        x=(struct cvor *)malloc(sizeof(*x));
        x–>kljuc=k;
        x–>l=z;
        x–>d=z;
        if (glava==z) glava=x;
        else
            if(k<f–>kljuc) f–>l=x;
                else f–>d=x;
        return x;
    }
    else return drvoinsertr(k,x);
```

```
}
struct cvor *drvoinserti(int k, struct cvor *x)
{
    struct cvor *f;
    z->kljuc = k;
    do {
        f=x;
        if(k<x->kljuc) x=x->l;
            else x=x->d;
    } while(x! =z);
    x=(struct cvor *)malloc(sizeof(*x));
    x->kljuc=k;
    x->l=z;
    x->d=z;
    if (glava==z) glava=x;
    else
        if(k<f->kljuc) f->l=x;
            else f->d=x;
    return x;
}
```

Novi čvor se uvek dodaje na list drveta (tj. između lista i fiktivnog čvora *z). Pokazivač f pamti pokazivač na čvor – prethodnik unetog čvora.

Unošenje elementa sa vrednošću ključa koja već postoji u drvetu vrši se desno od čvora koji je već u drvetu.

Prosečni broj koraka za pretraživanje u drvetu koje je izgrađeno uzastopnim umetanjem n slučajnih ključeva je proporcionalan sa $\ln n$. U najgorem slučaju (kada se ključevi unose redom, ili obrnutim redom), ova metoda nije ništa bolja od sekvencijalnog pretraživanja.

11

Algoritmi u posebnim oblastima

11.1 Matematička indukcija u dokazivanju korektnosti i izvođenju programa

Za razvoj i dokazivanje korektnosti algoritama može da se koristi jedna dosta opšta matematička metoda dokazivanja – princip matematičke indukcije. Naime, invarijanta petlje može se posmatrati i kao primena ovog principa.

Matematička indukcija je veoma moćna tehnika dokazivanja. Ne ulazeći u formalizam, opišimo ukratko kako se ona primenjuje.

Neka je T teorema koju želimo da dokažemo, i neka uključuje parametar n nad skupom prirodnih brojeva. Umesto da dokazujemo direktno da T važi za svako n, dokazujemo sledeća dva uslova:

1. T važi za $n = 1$

2. za svako $n >= 1$, ako T važi za n, onda T važi za $n + 1$.

Iz uslova 1. i 2. direktno se izvodi važenje T za $n = 2$, a odatle, prema uslovu 2. za $n = 3$, itd.

Dok se uslov 1. (tzv. **baza indukcije**) obično dokazuje trivijalno, dokazivanje uslova 2, mada ne mora biti jednostavno, obično je jednostavnije od dokazivanja teoreme T neposredno, jer koristi pretpostavku o važenju T za n. Ova pretpostavka se naziva *induktivna hipoteza*.

Trivijalan primer dokaza primenom principa matematičke indukcije je dokaz da je suma prvih n prirodnih brojeva, $S(n)$, jednaka $n \times (n + 1)/2$.

Indukcija se izvodi po n. Ako je $n = 1$, tvrđenje trivijalno važi. Pretpostavljamo da tvrđenje važi za n, tj. da je $S(n) = n \times (n + 1)/2$, i dokazujemo da ono važi i za $n + 1$, tj. da je $S(n + 1) = (n + 1) \times (n + 2)/2$. Prema uvedenoj

oznaci, $S(n+1) = S(n) + (n+1)$, a prema induktivnoj hipotezi je $S(n+1) = n \times (n+1)/2 + (n+1) = (n \times (n+1) + 2 \times (n+1))/2 = ((n+1) \times (n+2))/2$, čime je dokaz završen.

Princip matematičke indukcije ima i druge, ekvivalentne formulacije, koje počinju od 0 umesto od 1, koje induktivnu hipotezu formulišu za $n-1$ umesto za n, a dokaz izvode za n umesto za $n+1$, koje u uslovu 2. uključuju sve vrednosti manje ili jednake n umesto samo vrednost n, koje invertuju osnovni princip formulišući induktivnu hipotezu za n a dokazujući tvrđenje za $n-1$, itd.

Princip matematičke indukcije često je veoma pogodno primeniti u dokazivanju korektnosti algoritama, ali i u razvoju (izvođenju) algoritama iz induktivne hipoteze. Ukoliko algoritam sadrži ciklus (petlju), primenom matematičke indukcije po broju izvršavanja petlje može se dokazati da je rezultat izvršavanja petlje upravo onaj koji se očekuje, tj. korektan. Induktivna hipoteza tada postaje invarijanta petlje.

11.2 Numerički algoritmi

Pod numeričkim algoritmima podrazumevaju se algoritmi nad brojevima. Primer takvog algoritma je izračunavanje vrednosti polinoma.

Primer (izračunavanje vrednosti polinoma).

Neka je dat sledeći problem: za dati niz realnih brojeva, $a_n, a_{n-1}, \ldots, a_1, a_0$ $(n \geq 0)$, i realni broj x, izračunati vrednost polinoma $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$.

Pokažimo kako princip matematičke indukcije dovodi jednostavno do algoritma za rešavanje ovog problema, i to ne do jednog, već do većeg broja sve boljih rešenja.

Problem uključuje $n+2$ broja. Induktivni pristup rešavanju problema je da se on svede na manji problem. Prirodni korak je pokušaj da se problem svede na problem bez člana a_n, tj. da se nađe algoritam za izračunavanje vrednosti polinoma $P_{n-1}(x) = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0$. Ovo je isti problem, samo sa jednim parametrom manje. Zato pokušavamo da ga rešimo indukcijom.

Baza indukcije, izračunavanje a_0, je trivijalno. Pretpostavimo da znamo da izračunamo vrednost polinoma $P_{n-1}(x)$ (induktivna hipoteza), tada pokazujemo kako se izračunava vrednost polinoma $P_n(x)$: $P_n(x) = P_{n-1}(x) + a_n x^n$.

Algoritam se svodi na izračunavanje vrednosti polinoma zdesna ulevo (poznati su ceo broj $n \geq 0$, realni broj x, i niz koeficijenata polinoma $a_0, \ldots, a_n$):

```
{
    i ← 0;
    P_i(x) ← a_0;
    while (i < n) {
            i ← i + 1;
            P_i(x) ← P_{i-1}(x) + a_i x^i;
        }
}
```

Algoritam je očigledan, korektan, ali i neefikasan. On zahteva $n+(n-1)+(n-2)+\cdots+1 = n(n+1)/2$ množenja i n sabiranja.

Prvo poboljšanje algoritma je u smanjenju suvišnog računanja. Naime, stepen od x se računa od početka u svakom koraku, mada je moguće za računanje stepena x^n koristiti prethodno izračunati stepen x^{n-1}. Zato računanje stepena x^{n-1} uključujemo u induktivnu hipotezu koja sada glasi: znamo da izračunamo vrednost polinima $P_{n-1}(x)$, i znamo da izračunamo x^{n-1}. Sada je vrednost polinoma $P_n(x)$ jednaka $P_{n-1}(x) + a_n \times x \times x^{n-1}$, tj. izračunavanje vrednosti polinoma P_n zahteva poznavanje vrednosti polinoma P_{n-1}, stepena x^{n-1}, i dva množenja (jedno x sa x^{n-1} i drugo x^n sa a_n) i jedno sabiranje ($P_{n-1}(x)$ sa $a_n x^n$). Zato je za izračunavanje vrednosti polinoma P_n sada potrebno $2n$ množenja i n sabiranja. Algoritam sada ima sledeći oblik:

```
{
    i ← 0;
    P_i(x) ← a_0;
    xs ← 1; /* (xs je tekuća vrednost stepena od x) */
    while (i < n) {
            i ← i + 1;
            xs ← xs × x
            P_i(x) ← P_{i-1}(x) + a_i × xs;
        }
}
```

Pored ovog, dobrog algoritma, postoji još bolji. Naime, pored svođenja koje se sastoji u uklanjanju koeficijenta a_n, može se posmatrati i svođenje koje uklanja koeficijent a_0. Tada koeficijent a_n postaje $(n-1)$-vi koeficijent, a_{n-1} – $(n-2)$-gi, a a_1 – 0-ti, tj. svedeni polinom, koji će se razmatrati u induktivnoj hipotezi, je $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1}x^{n-2} + \cdots + a_1$. Sada se vrednost polinoma $P_n(x)$ izračunava kao $P_n(x) = x \times P'_{n-1}(x) + a_0$. Ovoga puta je za izračunavanje vrednosti polinoma $P_n(x)$ potrebno poznavanje vrednosti polinoma $P'_{n-1}(x)$ i jedno množenje i jedno sabiranje. Algoritam je poznat kao Hornerova šema (engl. matematičar W.G.Horner) i može se predstaviti formulom $P_n(x) = ((\ldots((a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0$, tj. sledećim nizom koraka:

```
{
    i ← 0;
    P'_i(x) ← a_n;
    while (i < n) {
            i ← i + 1;
            P'_i(x) ← x × P'_{i-1}(x) + a_{n-i}
        }
/* P_n(x) = P'_n(x) */
}
```

11.3 Nenumerički algoritmi

Nenumerički algoritmi odnose se na probleme koji u izvornom obliku nisu izraženi brojevima (mada se uvek mogu transformisati i u probleme nad brojevima). Takvi su algoritmi u teoriji grafova, u geometriji, teoriji formalnih reči, teoriji igara, itd. Navešćemo neke primere nenumeričkih algoritama u geometriji, nad niskama i grafovima.

11.4 Algoritmi u geometriji

Geometrijski algoritmi igraju značajnu ulogu u mnogim oblastima računarstva kao što su računarska grafika, projektovanje uz pomoć računara, robotika, baze podataka. Oblast koja proučava i implementira geometrijske algoritme obično se naziva *računarska geometrija* (engl. computational geometry). Osnovni dvodimenzionalni objekti kojima ovakvi algoritmi operišu jesu tačka, prava, segment prave, poligon, krug, itd. Za svaki objekat bira se prigodna reprezentacija, npr. tačka se može predstaviti parom koordinata u pravouglom Dekartovom sistemu. Analogno se predstavljaju prava, poluprava i duž (koordinatama dve tačke), odnosno poligon sa n temena (koordinatama tih temena). Putanja P sastoji se od niza tačaka $p_1, p_2, \ldots, p_n$ i niza duži $p_1 - p_2, p_2 - p_3, \ldots, p_{n-1} - p_n$ koje ih povezuju. Bitno je da je reč o nizovima a ne o skupovima, jer različiti redosledi tačaka proizvode različite poligone. Zatvorena putanja (*poligon*) je putanja čije se prva i poslednja tačka poklapaju. Tačke koje određuju poligon jesu temena poligona, a duži koje ih povezuju jesu stranice poligona. Poligon je *jednostavan* ako njegova putanja ne preseca samu sebe, tj. ako nijedne dve njegove stranice nemaju zajedničkih tačaka osim susednih stranica u zajedničkom temenu. Poligon je *konveksan* ako je svaka duž čije su krajnje tačke unutar poligona, takođe u celosti unutar tog poligona.

Jedna nezgoda sa geometrijskim algoritmima je postojanje velikog broja specijalnih slučajeva. Na primer, dve prave u ravni se obično seku u jednoj tački, ali se mogu i poklapati ili biti paralelne.

Primer.

Problem: Za dati jednostavni poligon P sa temenima $t_1, t_2, \ldots, t_n$ i stranicama $s_1, s_2, \ldots, s_n$, i datu tačku q sa koordinatama (x_0, y_0), odrediti da li se tačka nalazi unutar ili izvan poligona.

Poligon o kome je reč ne mora biti konveksan i može biti dosta složen. Crtanjem i analiziranjem i rešavanjem pojedinačnih zadataka ovog problema može se doći do zaključka (koji se u geometriji dokazuje) da je tačka q unutar poligona P ako i samo ako je broj presečnih tačaka bilo koje poluprave sa krajem u tački q sa stranicama poligona P neparan (slika 11.1).

Problemi na koje se nailazi pri izgradnji algoritma je da su tačke predstavljene parovima koordinata (a ne vizuelno kao na crtežu), i specijalni slučajevi kao što je poluprava na kojoj leži neka od stranica poligona, ili koja prolazi kroz neko od temena poligona. Ako se izuzmu takvi specijalni slučajevi (koji se razrešavaju dopunskim razmatranjem), i ako se umesto proizvoljne poluprave uzme baš poluprava

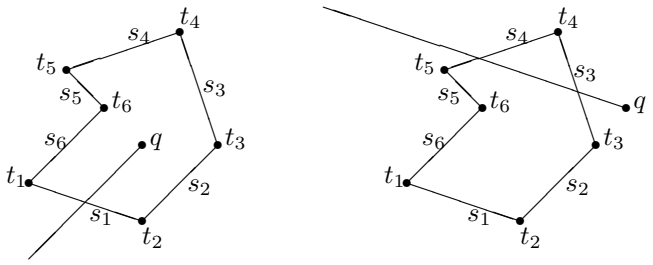


Figure 11.1: Odnos tačke i poligona – poluprava proizvoljnog položaja

kojoj je krajnja tačka q, koja je paralelna y-osi i nalazi se ispod tačke q (slika 11.2), algoritam može imati sledeći oblik:

1. br_pres_tačaka $\leftarrow 0$;
2. za svaku stranicu s_i poligona P uraditi:
   2.a. ako prava $x = x_0$ seče s_i onda
      2.a.a. neka je y_i y-koordinata preseka prave $x = x_0$ i stranice s_i;
      2.a.b. ako je $y_i < y_0$ onda
         2.a.b.a. uvećati br_pres_tačaka (za 1);
3. ako je br_pres_tačaka neparan onda je tačka u unutrašnjosti poligona, kraj; inače tačka nije unutar poligona P, kraj.

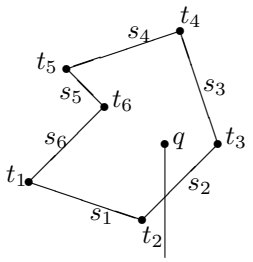


Figure 11.2: Odnos tačke i poligona – poluprava specijalnog položaja

11.5 Algoritmi sravnjivanja niski (pretraživanja nad niskama)

Niske (stringovi) su centralni tip podataka u sistemima za obradu reči i rad sa tekstovima. Kako te niske (slova, cifara, specijalnih znakova) mogu biti veoma

duge (npr. tekst jedne knjige je niska od preko milion karaktera), potrebni su veoma efikasni algoritmi za njihovu obradu.

Osnovna operacija nad niskama (tekstom) je *sravnjivanje* ili *podudaranje* niski: za dati tekst (nisku) T dužine N i dati obrazac (nisku) s dužine M, naći pojavljivanje (ili sva pojavljivanja) obrasca s u tekstu T.

11.5.1 Algoritam grube sile

U širokoj upotrebi je očigledni algoritam "grube sile" za problem sravnjivanja niski. Mada njegovo izvršavanje u najgorem slučaju zahteva vreme proporcionalno sa $M \times N$, većina niski (tekstova) u mnogim aplikacijama je takva da zahteva vreme proporcionalno sa $M + N$.

Ovaj algoritam, za svaku poziciju u tekstu na kojoj se može naći obrazac, proverava da li se na toj poziciji obrazac zaista i nalazi. Sledeća funkcija traži prvo pojavljivanje obrasca $s[1..M]$ u tekstu $T[1..N]$:

```
char t[10000];
char s[20];
int grubasila(int m, int n)
{
    int i=1, j=1;
    do {
        if(t[i] == s[j]) {
            i++; j++;
        }
        else {
            i=i – j+2; j=1;
        }
    while((j<=m) && (i<=n));
    if(j>m) return i–m; else return i;
}
```

Funkcija održava jedan brojač pozicija u tekstu (i) i drugi u obrascu (j). Sve dok se karakteri na tim pozicijama podudaraju, oba brojača se uvećavaju. Ako se dođe do kraja obrasca ($j > m$), nađeno je jedno pojavljivanje obrasca u tekstu. Ako se karakteri na pozicijama u tekstu i obrascu razlikuju, onda se j postavlja da ponovo pokazuje na početnu poziciju obrasca, a i se vraća na prvu poziciju u tekstu koja sledi za onom od koje je poslednje traženje obrasca završeno neuspešno. Ako se dođe do kraja teksta ($i > n$), nema podudaranja. Ako se obrazac ne nalazi u tekstu, funkcija vraća vrednost $n + 1$.

U aplikacijama editovanja teksta, unutrašnja petlja funkcije **grubasila** retko se ponavlja (retke su podniske teksta koje imaju zajednički početak sa obrascem duži od 1), pa je vreme izvršavanja skoro proporcionalno sa dužinom teksta (N). Međutim, kada su i tekst i obrazac – binarne niske (nad azbukom {0,1}), kao što je slučaj kod binarnog predstavljanja slike, program (funkcija) **grubasila** može da se izvršava znatno sporije. Zbog toga su potrebni i razvijeni efikasniji algoritmi.

11.5.2 Algoritam Knut-Moris-Prat-a

Osnovna ideja ovog algoritma je sledeća: kada se otkrije nepodudarnost karaktera u tekstu i obrascu, poznati su karakteri od kojih se sastoji naš pogrešni pokušaj sravnjivanja. Tu informaciju bi trebalo nekako iskoristiti da bi se izbeglo vraćanje brojača pozicija u tekstu, preko tih poznatih karaktera. Na primer, ako se početno slovo obrasca ne pojavljuje više ni jedanput u obrascu (npr. "n" u obrascu "nizovski", ili "1" u obrascu "1000000"), onda se, pri otkrivenom nepodudaranju (za neko i i j) sa karakterom u tekstu (npr. tekst "...nanize...", nepodudaranje je na karakteru "o"), nema potrebe za vraćanjem brojača i na karaktere podudarenih delova teksta i obrasca ("iz"), jer se među njima ne može naći novi početak obrasca; sa brojačem u tekstu može se nastaviti unapred (i pokazuje na "e"), a samo se brojač u obrascu vraća na početak, ($j = 1$, što odgovara opet slovu "n"). Dakle, iskaz $i = i - j + 2$ u funkciji **grubasila**, može se zameniti iskazom $i = i + 1$.

Mada je slučaj da se početno slovo obrasca ne pojavljuje više u obrascu – specijalan, pokazuje se da je Knut-Moris-Prat-ov algoritam njegovo uopštenje.

Na primer, ako je binarni tekst T: 1010100111..., a obrazac s: 10100111, prvo nepodudaranje će se otkriti na petom karakteru, ali se tada ne mora vratiti na drugi karakter teksta da bi se od njega tražio novi početak obrasca. Kako podudareni deo ("1010") sadrži završnu podnisku koja je jednaka početnoj ("10"), moguće je vratiti se na početak te završne podniske u tekstu (drugo "1", tj. treći karakter), i odatle tražiti obrazac ispočetka. S druge strane, kako obrazac znamo unapred, poznato nam je koji sve karakteri čine prepoznati deo teksta ("1010"), pa i njegov završni deo (drugo "10"). Zato je sa sravnjivanjem moguće početi od sledećeg karaktera teksta (petog, "1"), i karaktera koji sledi za prvom podniskom "10" u obrascu – trećeg karaktera, "1".

Ako se dati obrazac programski "ugradi" u funkciju sravnjivanja, za naš, specifični obrazac "10100111", funkcija za Knut-Moris-Prat-ov algoritam sravnjivanja niski, može da uključi sledeće iskaze:

```
      i=0;
0:    i=i+1;
1:    if (t[i] !='1') goto 0; i++;
2:    if(t[i] !='0') goto 1; i++;
3:    if(t[i] !='1') goto 1; i++;
4:    if(t[i] !='0') goto 2; i++;
5:    if (t[i] !='0') goto 3; i++;
6:    if (t[i] !='1') goto 1; i++;
7:    if(t[i] !='1') goto 2; i++;
8:    if (t[i] !='1') goto 2; i++;
      return i–8;
```

Ovi se iskazi izvršavaju sve dok se ne dođe do kraja teksta ili dok se ne naiđe na traženi obrazac (često se prvi uslov za kraj eliminiše time što se na kraj teksta dopiše obrazac).

Labele u prethodnom iskazu zavise od obrasca – u slučaju nepodudaranja na drugom, trećem ili šestom karakteru obrasca, nastavlja se, od tekućeg karaktera

teksta, traženje obrasca ispočetka, ako do nepodudaranja dođe na četvrtom, sedmom ili osmom karakteru obrasca (u tekstu je nađen početak obrasca 101, 101001 ili 1010011, ali se sledeći karakter u tekstu razlikuje), od tekućeg karaktera teksta nastavlja se sa poređenjem drugog karaktera obrasca (za prvi, "1", zna se da je jednak poslednjem karakteru teksta).

Za obrazac s u opštem slučaju, za svaku njegovu poziciju j treba izračunati na koju poziciju to j treba vratiti ako na njoj dođe do nepodudaranja sa tekstom. Neka je vrednost tog "pomaka", za poziciju j, $pomak[j]$. Tada je funkcija koja realizuje Knut-Moris-Prat-ov algoritam za opšti slučaj teksta i obrasca, sledećeg oblika:

```
int kmp_srav(int m, int n)
{
    int i=1, j=1;
    do {
        if((j==0) || (t[i]==s[j])) {
            i++; j++;
        }
        else { j= pomak[j];}
    while ((j<=m) && (i<=n));
    if (j>m) return i-m;
    else return i;
}
```

Sam niz pomak[j] izračunava se, u opštem slučaju, na sličan način, samo sada poređenjem obrasca sa samim sobom, da bi se identifikovale jednake podniske.

```
void initpomak(void)
{
    int i=1, j=0;
    pomak[1]=0;
    do {
        if((j==0) || (s[i]==s[j])) {
            i++; j++; pomak[i]=j; }
        else {j= pomak[j];}
    while (i<=m);
}
```

Za vežbu: Napisati program koji testira funkcije grubasila i kmp_srav.

11.6 Algoritmi nad grafovima: traganje unazad - bektreking

Za veliki broj problema nije moguće naći efikasan algoritam koji će se izvršiti u polinomijalnom vremenu od dimenzije problema, već postoje samo algoritmi eksponencijalne vremenske složenosti koji, po pravilu, uključuju tehniku traganja un-

azad (engl. backtracking). U slučaju problema nad grafovima, to znači da problem pokušavamo da rešimo prateći jednu moguću putanju koja polazi iz nekog čvora, ali ako ustanovimo, na nekom koraku, da tom putanjom ne možemo da rešimo problem, vraćamo se u neki prethodno posećeni čvor te putanje i pokušavamo kretanje nekom drugom putanjom iz tog čvora.

Primer: Razmotrimo problem bojenja grafa u tri boje. Dat je graf $\Gamma = (A, R)$, gde je A – konačni skup a R – binarna relacija na skupu A ($R \subseteq A \times A$, gde je $A \times A$ - Dekartov proizvod skupa A sa samim sobom, tj. $A \times A = \{(a, b) | a, b \in A\}$). Elemente skupa A zovemo *čvorovi* a elemente skupa R – *grane* grafa Γ. Graf Γ je neusmeren, što znači da je relacija R simetrična ($(a, b) \in R \equiv (b, a) \in R$). Čvor a je susedan čvoru b ako postoji grana $(a, b) \in R$. Problem: da li je moguće svaki čvor grafa obojiti jednom od tri zadate boje, tako da nijedna dva susedna čvora nemaju istu boju?

Ako graf ima n čvorova, a svaki čvor se može obojiti jednom od tri boje, postoji 3^n različitih načina da se svi čvorovi grafa oboje. Od tog broja, samo mali broj rešenja (ako ih uopšte ima) biće dobro (u skladu sa ograničenjem problema), osim ako graf nema grana uopšte. Jedno rešenje (algoritam) polazi od proizvoljna dva susedna čvora (ako takvi postoje) i njihovog bojenja proizvoljnim dvema različitim bojama. Zatim se bira čvor iz preostalog skupa neobojenih čvorova i on se boji bojom kojom nije obojen nijedan njegov sused, ako je to moguće. Poslednji korak se ponavlja dok je moguće, ili dok se ne dobije valjano obojeni graf. Ako na nekom koraku nije moguće obojiti izabrani čvor, vrši se traganje unazad, tako što se čvor obojen u prethodnom koraku pokuša obojiti nekom drugom bojom.

Algoritam ima sledeći oblik:

Algoritam **Bojenje-3**(Γ, U);
Ulaz: $\Gamma = (A, R)$ (neusmeren graf) i U (skup obojenuh čvorova i njihovih boja; na početku prazan); graf je konstantan, a U je promenljiva – vrednosni parametar algoritma (funkcije);
Izlaz: Dodela jedne od tri boje svakom čvoru od Γ.

```
{
    if(A == U) { "bojenje završeno"; kraj; }
    else
        {
            izabrati čvor a koji nije u U;
            for (C = 1; C <= 3;C++)
                if (nijedan sused od a nije obojen bojom C)
                {
                    dodati a skupu U sa bojom C;
                    Bojenje-3(Γ, U);
                }
        }
}
```

Sasvim je verovatno da će se ovaj algoritam izvršavati u eksponencijalnom broju koraka, što je čest slučaj kod algoritama sa traganjem unazad. Ipak, postoji nada da se, dobro odabranim redosledom čvorova i određenim heuristikama, u nekim slučajevima dobija bolje vreme izvršavanja.

12

Formalna definicija algoritma: Tjuringova mašina

Još apstraktnija definicija računskog procesa, u odnosu na Fon-Nojmanovu mašinu, predstavlja formalna definicija algoritma koju je formulisao engleski matematičar Alen Tjuring 1937. godine. Tjuringova mašina otkriva suštinu pojma algoritma razmatranjem postupaka ostvarivih na mašini. Način rada mašine prikazuje se jednom standardnom shemom, jednostavnom po logičkoj strukturi ali dovoljno preciznom da omogući razna matematička istraživanja.

Tjuringova mašina je definisana pre nastanka savremenih elektronskih računara. To je algoritamski sistem, konceptualna ("papirna", hipotetička- pretpostavljena) mašina, čija fizička realizacija ne postoji. Rad Tjuringove mašine (tj. postupak izračunavanja koji se odvija po njenoj definiciji) samo oponaša rad čoveka koji računa prema strogom propisu. Od računara se razlikuje u dva elementa:

-raščlanjavanje računskog postupka je dovedeno do granične mogućnosti, što produžuje sam postupak ali ga i standardizuje;

-"memorija" Tjuringove mašine je neograničena.

Neformalna definicija Tjuringove mašine uključuje nekoliko elemenata:

-beskonačna traka sa nizom polja (ćelija) ("memorija");

-konačni skup $A = \{s_0, s_1, \ldots, s_n\}$ znakova (simbola), koji se naziva spoljašnja azbuka mašine. U svakom trenutku, jedna ćelija beskonačne trake sadrži tačno jedan simbol spoljašnje azbuke. Jedan od znakova ima ulogu "praznog" znaka, npr. s_0, sa specifičnim značenjem: upis praznog znaka u neko polje "memorije" briše prethodno prisutni znak. Za sve ćelije za koje se drugačije ne naglasi, podrazumeva se da sadrže prazan znak;

-glava za čitanje i upis znakova spoljašnje azbuke;

-komandno-logički blok koji, izvršavanjem određenog programa, upravlja

pomeranjem glave i upisom na traku. Komandno logički blok upravlja mašinom i u raznim vremenskim trenucima nalazi se u različitim komandnim stanjima. Skup komandnih stanja je konačni skup $Q = \{q_0, q_1, \ldots, q_m\}$, u kome dva stanja imaju specifično značenje: jedno stanje izdvaja se kao početno stanje, npr. q_1, a drugo kao završno stanje, npr q_0. Komandno-logički blok sadrži i informaciju o pomeranju glave za čitanje i upis ulevo, udesno, ili nepomeranju, što se označava simbolima iz skupa $P = \{L, D, N\}$. Skup simbola $P \cup Q$ čini unutrašnju azbuku mašine.

Na početku rada Tjuringove mašine (u početnom stanju q_1), na traci je upisan početni izraz tj. početna informacija, koja odgovara ulaznim podacima u program ili algoritam. Način postavljanja početne informaciju na traku je nebitan i nije obuhvaćen definicijom Tjuringove mašine (to je moguće upravo zbog hipotetičkog karaktera mašine u kojoj se pojedini elementi mogu "pretpostaviti"). Rad mašine odvija se u taktovima. Takt se sastoji od čitanja znaka ispod kojeg je glava za čitanje, upisa novog znaka, eventualnog pomeranja glave za jedno mesto ulevo, udesno, i prelaza u novo stanje, u zavisnosti od pročitanog znaka i tekućeg stanja, pri čemu se informacija na traci transformiše u novu međuinformaciju (sadržaj trake posle jednog takta).

Rad mašine u svakom taktu prikazuje se naredbom oblika

$$q_i s_j \to s_k p q_l.$$

Naredba se može primeniti samo u trenutku kada je ispod glave za čitanje simbol s_j, a kontrolno-logički blok je u stanju q_i.

U taktu koji se odvija prema ovoj naredbi izvršava se sledeći niz radnji:
-čitanje slova s_j ispod kojeg je glava za čitanje;
-upis slova s_k u ćeliju ispod koje je glava za čitanje i upis;
-pomeranje glave u zavisnosti od znaka p;
-izmena komandnog stanja q_i u q_l;
-provera da li je nastupilo završno stanje (stanje zaustavljanja); ako nije, prelazi se na sledeći takt.

Niz naredbi Tjuringove mašine čini Tjuringov program. Komandno-logički blok, izvršavanjem tog programa, upravlja pomeranjem glave za čitanje i upis i upisom novih sadržaja na traku. Tako komandno-logički blok realizuje funkciju

$$Q \times A \to A \times P \times Q,$$

koja se naziva logička funkcija Tjuringove mašine. Ta funkcija se može predstaviti tabelarno, Tjuringovom funkcijskom shemom, oblika

	q_0	q_1	$\ldots$	q_i	$\ldots$	q_m
s_0						
s_1						
$\ldots$						
s_j				$s_k p q_l$		
$\ldots$						
s_n						

U tabeli su popunjena samo ona polja koja odgovaraju prelazima definisanim naredbama programa. To su polja koja se nalaze u preseku j -te vrste i i -te kolone, za svako i, j za koje postoji naredba u programu sa znakom s_j i stanjem q_i na levoj strani naredbe. Kolona za završno stanje q_0 se obično isključuje, s obzirom da u tom stanju nije moguć prelaz ni za jedan pročitani simbol spoljašnje azbuke.

U zavisnosti od početne informacije A, moguće je da:

a) posle konačnog broja taktova mašina stane u završnom stanju q_0 (na traci je zapisana informacija B). Mašina je u tom slučaju primenljiva na informaciju A i transformiše je u rezultat B;

b) mašina nikad ne stane (ne dođe u završno stanje q_0). Mašina u tom slučaju nije primenljiva na početnu informaciju A.

Kaže se da Tjuringova mašina rešava neku klasu zadataka ako je primenljiva na svaku informaciju koja izražava početne uslove bilo kog zadatka date klase, i transformiše tu informaciju u informaciju koja izražava rešenje tog zadatka.

Tjuringova mašina se može i formalno definisati na sledeći način:

Definicija 1 (Tjuringova mašina). Tjuringova mašina T je uređena trojka

$$< A, Q, Pr >,$$

gde je

1. A - spoljašnja azbuka
2. Q - unutrašnja azbuka
3. Pr - program, tj. konačni skup naredbi.

Naredba je uređena petorka

$$< q_i, s_j, s_k, p, q_l >,$$

gde je $q_i, q_l \in Q$, $s_j, s_k \in A$, $p \in \{L, N, D\}$, i označava se sa $q_i s_j \to s_k p q_l$. Leva strana naredbe je par $(q_i,\ s_j)$ (tj. $q_i s_j$), a desna strana naredbe je trojka $(s_k,\ p,\ q_l)$ (tj. $s_k p q_l$). U programu se ne smeju naći dve naredbe sa jednakim levim stranama (deterministička Tjuringova mašina).

Primer 1. Napisati Tjuringovu mašinu koja nenegativni ceo broj x uvećava za 1.

Uzmimo za spoljašnju azbuku skup $A = \{0, 1\}$, gde je znak 0 prazan znak. To je skup čijim znakovima (i samo njima) će biti zapisana početna informacija, međuinformacije i rezultat. Neka se ceo nenegativni broj x u ovoj azbuci predstavlja pomoću $x+1$ jedinice (da bi i ceo broj 0 imao reprezentaciju različitu od praznog znaka). Dakle, potrebno je napisati Tjuringovu mašinu koja će realizovati funkciju $f_1(x) = x + 1$, tj. Tjuringovu mašinu T_{f_1} koja realizuje preslikavanje T_{f_1} : $q_1 1^{x+1} \to q_0 1^{x+2}$ (oznaka 1^{x+1} predstavlja reč sastavljenu od $x + 1$ jedinice; analogno 1^{x+2}). Grafički se ovo preslikavanje za $x = 2$ može predstaviti na sledeći način:

$$\ldots 0011100\ldots \qquad \rightarrow \ldots 00111100\ldots$$

Program se može realizovati tako da, polazeći od krajnje leve jedinice u zapisu broja x , pomera glavu za čitanje za jedno mesto ulevo, nulu na koju tamo naiđe zamenjuje jedinicom i stane. Ovakav program sastoji se od dve naredbe, tj.

Pr = {
$q_1 1 \rightarrow 1Lq_2$ (pomeranje glave za jedno mesto ulevo od krajnje leve jedinice)
$q_2 0 \rightarrow 1Nq_0$ (zamena nule levo od krajnje leve jedinice, jedinicom i kraj)}.

Program definiše i unutrašnju azbuku $\{q_0, q_1, q_2\}$, čime su definisani svi elementi Tjuringove mašine T_{f_1} (skupovi A, Q, Pr).

Odgovarajuća Tjuringova funkcijska shema ima oblik

	q_1	q_2
0		$1Nq_0$
1	$1Lq_2$	

Niz konfiguracija kroz koje Tjuringova mašina T_{f_1} prolazi u svom radu je sledeći:

$q_1 1^{x+1}$ {početna konfiguracija}
$q_2 01^{x+1}$
$q_0 1^{x+2}$ {završna konfiguracija}

Relativno jednostavni algoritmi (npr. dodavanje jedinice, oduzimanje, množenje, računanje apsolutne vrednosti razlike) jednostavno se ostvaruju na Tjuringovoj mašini. Za složenije algoritme ovakav postupak je vrlo kompleksan. Stoga je opšti princip pri izgradnji Tjuringovog programa - koristiti prethodno sastavljene programe kao posebne blokove novog programa. Za to se koriste postupci serijske kompozicije, grananja i ponavljanja Tjuringovih programa. Dokazuje se da je izražajna moć formalizma Tjuringove mašine jednaka izražajnoj moći Fon-Nojmanove mašine, pa i savremenih elektronskih računara. Zato je formulisana i hipoteza (tzv. Čerčova hipoteza) o ekvivalentnosti Tjuringove mašine i intuitivnog pojma algoritma, tj. o tome da se Tjuringovom mašinom može zapisati svaki algoritam tj. program (koji se može zamisliti). Ova se hipoteza ne može dokazati jer uspostavlja odnos ekvivalencije između formalnog pojma (Tjuringove mašine) i neformalnog pojma (algoritma), ali joj u prilog ide činjenica da ne postoji kontraprimer.

13

Algoritamski nerešivi problemi

13.1 Potreba za formalizacijom pojma algoritma

Svaki postupak čije pojedine korake uzastopno izvršava automatska računarska mašina može se opisati algoritmom. S druge strane, svaki do danas poznati algoritam, kao i oni koji se mogu predvideti, načelno je moguće izvršiti na računaru. "Načelna" mogućnost odnosi se na činjenicu da izvršenje algoritma može biti proizvoljno dugo i da zahteva neograničeno duge zapise, dakle odnosi se na ograničenja prostora i vremena koja se uzimaju u obzir pri izvršavanju algoritma na računaru. Zato su neki (postojeći) algoritmi, ako su velike vremenske ili prostorne složenosti, praktično neostvarljivi na savremenom računaru (npr. algoritam najbolje – pobedničke ili remi – strategije u šahu). Zato se pod mogućnošću izvršenja algoritma na računaru pretpostavlja neograničena memorija.

Tačna matematička definicija pojma algoritma (pa i odgovarajuća definicija automatske računske mašine) razrađena je tek tridesetih godina 20. veka. Dakle, u toku dugih prošlih vekova matematičari nisu osećali potrebu za formalnom definicijom tog pojma, jer se reč algoritam pojavljivala samo u vezi sa izgradnjom konkretnih algoritama, a opis konstruisanog algoritma bio je dovoljan dokaz njegovog postojanja. Ali, sa razvojem matematike dolazilo se do sve težih problema, i do potrebe da se konstruišu algoritmi za sve šire klase zadataka.

Vrhunac u tom širenju klasa zadataka za koje su traženi algoritmi je problem pronalaženja algoritma koji bi rešio bilo koji matematički problem. Ovaj problem je prvi postavio nemački matematičar i filozof Gotfrid Vilhelm Lajbnic (Gottfried Wilhelm Leibniz) još u 17. veku. Mada nije uspeo da ga reši, Lajbnic je bio uveren da će takav algoritam, kada se matematika dovoljno razvije, biti razvijen.

Ovaj problem kasnije je preciziran kao jedan od najvažnijih problema matematike, i to kao problem prepoznavanja izvodljivosti.

Za mnoge od tih teških problema algoritmi dugo nisu nalaženi, pa je počelo da se sumnja da će ikada biti i pronađeni, tj. da uopšte i postoje. Međutim, za razliku od "dokaza" da algoritam za neki problem postoji – tako što se ponudi opis takvog algoritma, da bi se dokazalo da algoritam ne postoji neophodno je imati preciznu definiciju onoga što se dokazuje da ne postoji, tj. algoritma. Upravo je potreba za eventualnim dokazivanjem nepostojanja algoritma za neku klasu zadataka dovela do formalne definicije tog pojma.

Tridesetih godina 20. veka pojavio se veći broj formalnih definicija pojma algoritma. Jedna od tih definicija je i Tjuringova mašina (A. Tjuring, 1937.g). Savremena teorija algoritama nudi sledeću hipotezu o opštosti Tjuringove mašine kao algoritamskog sistema, tj. formalne definicije algoritma:

Svaki algoritam (u intuitivnom smislu tog pojma) može se zadati u obliku Tjuringove funkcijske sheme i realizovati odgovarajućom Tjuringovom mašinom.

Ova hipoteza je poznata kao **osnovna hipoteza teorije algoritama**, ili Čerčova teza. Ona se ne može dokazati jer uspostavlja ekvivalenciju između jednog formalnog pojma (Tjuringove mašine) i jednog neformalnog pojma (algoritma u intuitivnom smislu), ali joj u prilog ide činjenica da nije do danas nađen kontraprimer, kao i da su sve formalne definicije pojma algoritma (Tjuringova mašina, Postova mašina, Markovljevi (normalni) algoritmi, itd.) među sobom ekvivalentne. Poslednje tvrđenje se i formalno dokazuje.

Polazeći od neke formalne definicije pojma algoritma, npr. od definicije Tjuringove mašine, sada je jasno šta znači dokazati da je neki problem algoritamski nerešiv: to znači dokazati da ne postoji Tjuringova mašina (prema formalnoj definiciji tog pojma) koja rešava taj problem, tj. svaki zadatak iz klase koja određuje taj problem.

Navedimo primere nekih klasa zadataka koji su algoritamski nerešivi, tj. za koje ne postoje algoritmi koji bi svaki od zadataka tih klasa rešili.

13.2 Diofantske jednačine

Diofantska jednačina je jednačina oblika $P = 0$, gde je P polinom s celobrojnim koeficijentima i proizvoljnim brojem nepoznatih. Na primer, diofantske su jednačine

$x^2 + y^2 - z^2 = 0$,

$6x^{18} - x + 3 = 0$.

Diofantska jenačina može, a ne mora, imati celobrojna rešenja. Tako prva od prethodnih jenačina ima celobrojna rešenja (npr. $x = 3, y = 4, z = 5$), dok druga nema, jer za svaki ceo broj x važi $6x^{18} > x - 3$.

Na Svetskom kongresu matematičara u Parizu 1901. godine poznati nemački matematičar David Hilbert predstavio je 20 teških (nerešenih) problema. Jedan od tih problema (10. Hilbertov problem) odnosio se upravo na diofantske jednačine: treba izgraditi algoritam koji za proizvoljnu diofantsku jednačinu utvrđuje da li ona ima ili nema celobrojna rešenja.

U specijalnom slučaju diofantskih jednačina s jednom nepoznatom, takav algoritam je odavno poznat: ako jednačina

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$$

sa celobrojnim koeficijentima ima celobrojno rešenje x_0, onda je x_0 činilac slobodnog člana a_0. Zato algoritam u ovom slučaju može imati sledeći oblik:

1) naći sve činioce celog broja a_0 (takvih činilaca je konačno mnogo);

2) za svaki činilac (jedan po jedan), izračunati levu stranu jednačine stavljajući da je x jednako tom činiocu;

3) ako je za neki od činilaca u prethodnom koraku leva strana jednačine dobila vrednost 0, onda je taj činilac – celobrojno rešenje jednačine; ako to ne važi ni za jedan činilac od a_0, jednačina nema celobrojnih rešenja.

Hilbertov problem je dugo bio u centru interesovanja mnogih matematičara. Ne samo da nije nađen algoritam u opštem slučaju, kada je dve ili više nepoznatih, već je 1969. godine lenjingradski matematičar Matijasevič (Yuri Vladimirovich Matiyasevich) dokazao da takav algoritam ne postoji.

13.3 Nerešivost problema algoritmima posebnog oblika

Da bi se u opštem slučaju dokazalo nepostojanje algoritma za rešavanje zadatog problema, bilo je potrebno imati preciznu definiciju pojma algoritma. U nekim posebnim slučajevima, bezuspešno su tražena rešenja nekih problema algoritmima specifičnog oblika. Na primer, još su stari Grci postavili problem trisekcije ugla pomoću lenjira i šestara (po analogiji sa bisekcijom ugla). Ili, problem nalaženja rešenja jednačine n-tog stepena (za $n >= 5$) kao funkcije njenih koeficijenata, primenom osnovnih aritmetičkih operacija i korenovanja.

Dokazano je da ovakve, specifične metode za rešavanje ovih problema ne postoje, ali je prethodno bilo neophodno tačno definisati te objekte (metode) za koje se dokazuje da ne postoje. Te definicije, "konstrukcija lenjirom i šestarom", "rešenje pomoću korena", preciziraju smisao nekih algoritama posebnog oblika.

14

Strukturno programiranje

Jedan značajan, disciplinovan stil u programiranju, koji je dobro primenjivati i u programskom jeziku C, jeste *strukturno programiranje*.

Strukturno programiranje je sistematičan pristup razvoju korektnih i razumljivih programa. Strukturno programiranje je nastalo iz diskusije o prednostima konstrukcija za tok upravljanja u programu, koju je inicirao članak E. Dijkstra pod naslovom "GOTO iskaz se smatra štetnim" (engl. Go to statement considered harmful, 1968.). E. Dijkstra je još 1965. godine objavio izveštaj o tome da je kvalitet programera obrnuto proporcionalan broju GOTO iskaza u njihovim programima. GOTO iskaz eksplicitno šalje upravljanje na navedenu tačku u programu, pa zbog toga, prema Dijkstra, "poziva i navodi programera da napravi zbrku u programu". Umesto takvog iskaza, Dijkstra predlaže programski stil koji kombinuje sledeće dve ideje:

1. Strukturni tok upravljanja. Program je *struktuiran* ako je njegov tok upravljanja očigledan iz sintaksne strukture programskog teksta.

2. Invarijante. Invarijanta u tački p programa je tvrđenje (logički uslov o stanju izračunavanja) koje važi svaki put kada upravljanje dostigne tačku p (na primer, $x \leq y$).

Većina proceduralnih jezika, pa i programski jezik C, poseduje sledeće metode struktuiranja iskaza:

1. Serijska kompozicija (sekvenca). Ako su $S_1, S_2, \ldots, S_k$ ($k \geq 0$) iskazi, onda je njihova serijska kompozicija – lista iskaza koja se može zapisati $S_1; S_2; \ldots; S_k$ (ili u C-u, $\{S_1; S_2; \ldots; S_k;\}$).

2. Selekcija – uslovna struktura. Ako je E logički izraz a S_1, S_2 – iskazi (tj. liste iskaza), onda je iskaz selekcije (uslovni iskaz) oblika

   IF (E) S_1; ELSE S_2;

3. Iteracija – WHILE-petlja. Ako je E logički izraz a S iskaz (lista iskaza), onda je iteracija (WHILE-petlja) oblika

   WHILE (E) S;

Izbor ovih metoda struktuiranja iskaza u skladu je sa jednim rezultatom iz 1966. godine (C. Bohm, G. Jacopini), poznatim kao *teorema o struktuiranju*. Ova teorema tvrdi (i formalno se dokazuje) da svaki dijagram toka upravljanja (engl. flowchart) – drugim rečima svaki program, može da se transformiše u dijagram bez skoka (goto), tj. u dijagram sa jednim ulazom i jednim izlazom koji je, osim iskaza dodele, komponovan od samo 3 osnovna dijagrama: sekvence, selekcije (uslovnog iskaza) i iteracije (WHILE-petlje). Ovi osnovni dijagrami (osnovne strukture) prikazani su na slici 14.1.

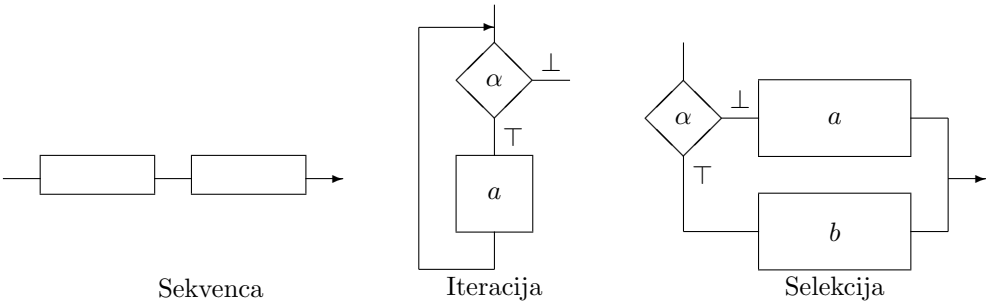


Figure 14.1: Osnovne upravljačke strukture

Autori pokazuju da važi i više, tj. da se i bez selekcije može izraziti svaki dijagram toka upravljanja (jer se selekcija može izraziti preko sekvence i iteracije, uz uvođenje izvesnog broja logičkih promenljivih i dodela).

Metodologija (stil) strukturnog programiranja nije u primeni teoreme o struktuiranju. Naime, to što se svaki (bilo kako napisani) program može na ovaj način naknadno "struktuirati", ne može učiniti (inicijalno) loše napisan program ni malo čitljivijim i razumljivijim. Šta više, tako "dostruktuirani" program može višestruko (čak eksponencijalno) da poraste u odnosu na svoj original.

Teorema o struktuiranju se, do duše, može primeniti u obrnutom smeru, naime, da se koraci globalne strukture programa zamenjuju strukturnim iskazima, i da se tako postiže "profinjenje" i modularnost, ili razvoj programa odozgo naniže (engl. top down), što već jeste jedan od principa strukturnog programiranja (uz disciplinu koja podrazumeva izbegavanje trikova i dobru dokumentaciju).

Sistematičan razvoj korektnog programa može da bude vođen invarijantama. Naime, invarijante pomažu da se statična struktura izvornog teksta programa dovede u vezu sa dinamičnošću izvršavanja programa odnosno ponašanja programa u vreme izvršavanja. Invarijante opisuju istovetnost odnosa među programskim

promenljivim pri raznim prolazima kroz iste tačke programa u toku izvršavanja programa. Dobro uočena invarijanta (i tačna i korisna) može da vodi top-down razvoj programa.

Ilustrujmo primenu invarijante u razvoju programa za serijsko pretraživanje niza.

Dat je niz a od n elemenata (> 0) (kao i ranije). Program treba da pronađe najmanji indeks i za koji je element $a[i]$ jednak zadatoj vrednosti x, ako takav postoji, inače program treba da vrati vrednost 0. Prvo rešenje ovog jednostavnog problema je sledeće:

poći od prvog elementa;
while (ima još elemenata)
    if (taj element je x)
        vratiti njegov indeks;
    else
        posmatrati sledeći element;
nije nađen, vratiti 0.

Programski fragment koji realizuje ovo rešenje je sledeći:

```
{
i = 1;
while (i <= n)
    if (x == a[i])
        return i;
    else i = i + 1;
return 0;
}
```

Ovo rešenje je moguće popraviti tako da ima samo jedan izlaz (umesto dva return).

Svaka iteracija vrši dva testa: prvi test, $i <= n$ zaustavlja pretraživanje kada nema više elemenata da se pretražuju; drugi test, $x == a[i]$, proverava da li je x nađeno na poziciji i.

Sledeće rešenje objedinjuje ova dva kriterijuma tako što uvodi $n+1$-vi element koji je jednak x: $a[n+1] = x$. Sada je pretraživanje uspešno ako se x nađe na poziciji i, $1 \leq i \leq n$, i neuspešno ako je $a[n+1] == x$. Ovde pomažu invarijante.

Postuslov (koji treba da važi posle pretraživanja) je:

/* (x nije u nizu a) ili (prvo x je $a[i]$ i $1 \leq i \leq n$) */

U oba slučaja imamo da je $x == a[i]$ i da x nije u intervalu $a[1..i-1]$ (invarijanta; $a[l..m]$ je oznaka za podniz $a[l], a[l+1], \ldots, a[m]$ u slučaju $l \leq m$, inače je to prazan skup). Razlika u ova dva slučaja je u vrednosti od i. Ako x nije u nizu, onda je $i == n+1$, inače je $1 \leq i < n+1$.

Sada se prethodni postupak može "doterati" tako da ima jedan izlaz i da važi prethodna invarijanta:

inicijalizacija;
pretražiti niz;

/*(x nije u nizu) ili (prvo x je $a[i]$ AND $1 \leq i < n+1$)*/
return i;

Ako se inicijalizacija eksplicira, invarijanta precizira a pretraživanje realizuje while-petljom, pretraživanje se završava ako nađemo x ili ako prođemo kroz svih n elemenata, tj. ako je $x == a[i]$ ili $i == n+1$. Iz postuslova, oba ova slučaja se svode na $x == a[i]$. Dakle, u petlji ostajemo sve dok je $x \neq a[i]$, pa fragment programa dobija sledeći konačni izgled:

```
{
a[n + 1] = x;
i = 1;
/*x ∉ a[1..i − 1] */
while (x ≠ a[i])
{
    i = i + 1;
    /* x ∉ a[1..i − 1] */
}
/* x == a[i] i x ∉ a[1..i − 1] i 1 ≤ i ≤ n + 1 */
return i;
}
```

Važna ideja uključena u metodologiju strukturnog programiranja i top-down razvoj programa jeste *modularnost*. Ilustrujmo ovu ideju primerom programa za sortiranje niza celih brojeva. Program uključuje više globalnih koraka, kao što su učitavanje elemenata niza, sortiranje i izdavanje sortiranog niza.

Dakle, globalna struktura programa može se sastojati iz sledeća tri modula:

učitati niz a sa n elemenata;
sortirati niz a;
izdati sortirani niz a;

Sada je potrebno izvršiti "profinjavanje" tj. projektovanje pojedinačnih modula. Pre svega, učitavanje uključuje i neki oblik sumeđe (interfejsa) sa korisnikom u obliku poruka i pitanja. Dalje, učitavanje niza može biti uspešno (ako poštuje ograničenje maksimalnog broja elemenata) ili neuspešno. Zato se učitavanje niza može realizovati sledećom funkcijom:

```
void ucitniz(int a[ ], int *np, int *uspp)
/* sva tri parametra su pokazivači jer funkcija menja vrednosti na koje oni
pokazuju */
{
    int i;
    printf("uneti broj elemenata niza ");
    scanf("%d", np);
    if ((*np > MAXN) || (*np < 1)) *uspp=0;
    else
        {
```

```
                *uspp=1;
                for (i=1; i<=(*np); i++)
                    scanf("%d", &a[i]);
            }
}
```

Program sada može da ima sledeću globalnu strukturu:

```
#include <stdio.h>
#define MAXN 100
int a[MAXN];
int n, uspesan;
void ucitniz(int [ ], int *, int *);
void (...); /* sortiranje niza a */
void (...); /* izdavanje sortiranog niza */
main()
{
    ucitniz(a, &n, &uspesan);
    if (uspesan) {
            sortirati niz a;
            izdati sortirani niz a;
        }
    else printf("lose unet broj elemenata\n");
}
```

Modul za sortiranje niza a može biti izgrađen metodom izbora najmanjeg elementa, tj. vođen idejom nalaženja najmanjeg elementa (u podnizu dužine n, n-1, ..., 2) i njegove razmene sa elementom na 1., 2., ..., n-1. mestu, kao što je izloženo ranije kod metode sortiranja izborom. Zato ovaj modul može da bude deklarisan sledećom funkcijom:

```
void sortiraj(int a[ ], int n)
{
    int i,min;
    for(i=1; i<n; i++) {
        min=najmanji(a,i,n);
        razmeni(a,&i,&min);
        }
}
```

Da bi se potpuno razvila funkcija sortiraj potreban je sledeći stepen profinjenja, tj. razviti funkcije najmanji i razmeni:

```
int najmanji(int a[ ], int i, int n)
{
    int j;
    int min=i;
    for (j=i+1; j<=n; j++)
        if (a[j] < a[min]) min=j;
```

```
    return min;
}
```

Funkciju za razmenu dva elementa moguće je realizovati različitim postupcima. Jedan način je ciklična razmena vrednosti elemenata uz pomoć trećeg – pomoćnog elementa istog tipa: t=a[min], a[min]=a[i], a[i]=t. Drugi način je direktna razmena dva elementa (na različitim lokacijama – u našem slučaju različitim indeksima) i ova ideja je realizovana u sledećoj deklaraciji funkcije "razmeni":

```
void razmeni(int a[ ], int *i, int *min)
{
    if ((*i)! = (*min)) {
            a[*i]=a[*i]+a[*min];
            a[*min]=a[*i]-a[*min];
            a[*i]=a[*i]-a[*min];
    }
}
```

Najzad, modul za izdavanje sortiranog niza elemenata može se realizovati funkcijom koja štampa jedan element u jednom redu izlazne datoteke:

```
void stampa(int a[ ], int n)
{
    int i;
    for (i=1; i<=n; i++)
        printf("%d\n", a[i]);
}
```

Razvijeni program sada ima sledeći oblik:

```
/* ucitavanje, sortiranje i stampanje sortiranog niza */
#include <stdio.h>
#define MAXN 100
int a[MAXN];
int n, uspesan;
void ucitniz(int [ ], int *, int *);
void sortiraj(int [ ], int );
void stampa(int [ ], int );
int najmanji(int [ ], int , int );
void razmeni(int [ ], int *, int *);
main()
{
    ucitniz(a,&n,&uspesan);
    if (uspesan) {
        sortiraj(a,n);
        stampa(a,n);
        }
    else printf("lose unet broj elemenata\n");
}
```

```
void ucitniz(int a[ ], int *np, int *uspp)
/* sva tri parametra su pokazivači jer funkcija menja vrednosti na koje oni
pokazuju */
{
    int i;
    printf("uneti broj elemenata niza ");
    scanf("%d", np);
    if ((*np > MAXN) || (*np < 1)) *uspp=0;
    else
        {
            *uspp=1;
            for (i=1; i<=(*np); i++)
                scanf("%d", &a[i]);
        }
}
void sortiraj(int a[ ], int n)
{
    int i,min;
    for(i=1; i<n; i++) {
        min=najmanji(a,i,n);
        razmeni(a,&i,&min);
    }
}
int najmanji(int a[ ], int i, int n)
{
    int j;
    int min=i;
    for (j=i+1; j<=n; j++)
        if (a[j] < a[min]) min=j;
    return min;
}
void razmeni(int a[ ], int *i, int *min)
{
    if ((*i)! = (*min)) {
            a[*i]=a[*i]+a[*min];
            a[*min]=a[*i]-a[*min];
            a[*i]=a[*i]-a[*min];
    }
}
void stampa(int a[ ], int n)
{
    int i;
    for (i=1; i<=n; i++) printf("%d\n", a[i]);
}
```

15

Klasifikacija programskih jezika. Programske paradigme

Programska paradigma je suštinski način na koji se grade elementi i struktura računarskog programa. To je pogled koji programer ima na program i njegovo izvravanje. Postoji veći broj različitih programskih paradigmi kao što su imperativna (proceduralna), deklarativna (funkcionalna, logička, problemski-orijenitsana), objektno-orijentisana, itd. Na primer, imperativna (proceduralna) paradigma podrazumeva pogled na program kao uređeni niz naredbi (eventualno sa izdvojenim procedurama/ funkcijama); objektno-orijentisana pardigma je zasnovana na skupu objekata koji dejstvuju međusobno. Po pravilu, veći broj programskih jezika podržava mehanizme i principe jedne programske paradigme, pa se takva klasa jezika često izjednačava sa tom programskom paradigmom. Tako se jedan pristup klasifikaciji programskih jezika zasniva baš na programskoj paradigmi kojoj pripadaju.

15.1 Klasifikacija programskih jezika

Jedna moguća globalna klasifikacija programskih jezika deli jezike na *mašinski zavisne* i *mašinski nezavisne* programske jezike.

Mašinski zavisni jezici su tesno vezani za konkretnu mašinu i uzimaju u obzir specifičnosti arhitekture odgovarajućeg procesora. To su jezici niskog nivoa. U mašinski zavisne jezike spadaju mašinski, simbolički i makro jezici.

Mašinski nezavisni programski jezici su programski jezici visokog nivoa, koji se grade nezavisno od računara na kom će se izvršavati. Oni su sredstvo u kome programer zapisuje svoje programerske ideje ne upuštajući se u detalje arhitekture računara. Mašinski nezavisni jezici su namenjeni primenama u različitim delatnos-

tima ljudi, i za svaku od tih primena grade se posebne klase programskih jezika visokog nivoa.

15.1.1 Mašinski zavisni jezici

Mašinski jezici. Mašinski jezik je jezik koji konkretna mašina (računar), zajedno sa svojim operativnim sistemom, "razume" bez pomoći druge programske opreme. Programi napisani na mašinskom jeziku jedne mašine mogu se izvršavati na toj mašini direktno, bez prethodne obrade. Svaki računar ima svoj mašinski jezik.

Sve informacije u mašinskom jeziku su binarne (to su nizovi 0 i 1), jer elektronske i magnetne komponente od kojih su sastavljeni glavni delovi računara mogu da registruju 2 diskretna stanja – npr. protiče/ne protiče struje, ili pozitivno/negativno namagnetisanje.

Na primer, zapis cifara dekadnog sistema binarnom azbukom je 0 (0), 1 (1), 01 (2), 11 (3), 100 (4), 101 (5), 110 (6), 111 (7), 1000 (8), 1001 (9).

U jednom memorijskom registru mogu se zapisati različite varijacije 0 i 1 koje odgovaraju različitim podacima ili instrukcijama.

Instrukcije mašinskog jezika sastoje se od binarnog kôda operacije i adrese operanada. Na primer, instrukcija

0110001110010100

može da predstavlja instrukciju sabiranja (kôd 0110) operanada koji su na adresama 001110 i 010100.

Ovaj niz 0 i 1 može da predstavlja i podatak. Ako se, na primer, u registru sa adresom 001110 ($(13)_{10}$) nađe taj isti sadržaj (0110001110010100), onda će on biti protumačen kao broj $2^2 + 2^4 + 2^7 + 2^8 + 2^9 + 2^{13} + 2^{14}$ ($(25492)_{10}$), i sabraće se sa sadržajem registra sa adrese 010100.

Da li će se sadržaj jednog registra interpretirati kao instrukcija ili kao podatak zavisi od adrese tog registra. Naime, program na mašinskom jeziku unosi se u memoriju računara od tačno određene adrese, i njegovo izvršavanje počinje od te adrese (sadržaj te i narednih adresa interpretiraju se kao instrukcije). Sadržaj registra sa adresom jednakom adresi operanda neke instrukcije interpretira se kao podatak.

Programiranje na mašinskom jeziku omogućilo bi maksimalno korišćenje mogućnosti računara, ali njegova binarna forma čini mašinski jezik praktično neupotrebljivim za čoveka.

Svaki računar ima svoj mašinski jezik koji zavisi od mašine, od izgleda i dužine registra, od načina adresiranja, itd. Mašinski jezik se (osim na samom početku razvoja programiranja, 40-tih godina prošlog veka), nije koristio i ne koristi za programiranje, već samo za razumevanje procesa koji se u mašini dešavaju.

Simbolički jezici. Simbolički jezici, ili asembleri, uvedeni su da bi se omogućilo maksimalno korišćenje mogućnosti računara a pojednostavilo programiranje za čoveka. Simbolički jezik je uvek vezan za odgovarajući mašinski jezik. Instrukcije odgovaraju mašinskim instrukcijama, ali je zapis pogodniji – kôd operacije

zamenjen je mnemoničkom skraćenicom (npr. SAB umesto 0110), a binarne adrese zamenjene su simboličkim adresama (npr. a, b umesto 001110, 010100, redom). Tako bi prethodna mašinska instrukcija u simboličkom jezilu te mašine imala zapis SAB a, b.

Program na simboličkom jezilu ne može se direktno izvršavati na računaru već se mora prvo *prevesti* na mašinski jezik nekim jezičkim procesorom (programom).

Makro jezici. Ovi jezici predstavljaju proširenja simboličkih jezika *makro naredbama*. To su imenovane grupe naredbi simboličkog jezika koje se izdvajaju u posebne celine, a iz glavnog programa, gde je to potrebno, pozivaju se po imenu (sa mogućim parametrima). Makro naredbe skraćuju zapis programa, omogućuju jednostavnije i pouzdanije programiranje, ali ne smanjuju zauzeće memorije, jer se, pri prevođenju, (za razliku od procedura u višim programskim jezicima), prevedeni oblik makro naredbe "umeće" u glavni program gdegod je makro naredba pozvana.

Makro jezici i simbolički jezici se primenjuju za pisanje manjih modula sistemskih programa i biblioteka programa koji treba da budu optimalni u pogledu brzine izvršavanja i zauzimanja memorijskog prostora.

Osnovni nedostataci mašinski zavisnih jezika su, osim neudobnosti programiranja u njima, to što zahtevaju poznavanje tehničkih karakteristika računara i što su programi neprenosivi.

15.1.2 Mašinski nezavisni jezici

Zajednička karakteristika mašinski nezavisnih jezika je da se pri njihovoj izgradnji ne vodi računa o arhitekturi računara na kome će se koristiti, već o problematici kojoj su namenjeni (npr. u prirodnim naukama i matematici Fotran, Algol, Java, Lisp, Python, C, C++, Mathlab, u modeliranju i simulaciji GPSS, Simula, u sistemskom programiranju C, u obrazovanju Pascal, itd). To su jezici visokog nivoa čija struktura im omogućuje korišćenje na različitim računarima.

Za svaki mašinski nezavisni jezik neophodno je izgraditi jezički procesor – program koji će prevesti program sa ovog jezika na mašinski jezik "razumljiv" računaru.

Osnovne zajedničke karakteristike mašinski nezavisnih jezika su njihova (relativna, u odnosu na mašinski zavisne jezike) *jednostavnost*, *prirodnost*, *prenosivost* i *pouzdanost*. *Standardizacijom* ovih jezika ukidaju se nepotrebne razlike i omogućuje prenosivost programske opreme sa jednog računara na drugi.

Jedna klasifikacija mašinski nezavisnih jezika vezana je za njihovu programsku paradigmu. Važna klasa mašinski nezavisnih jezika su *imperativni (proceduralni, algoritamski)* programski jezici, koji su pogodni za predstavljanje algoritama kao serijske kompozicije koraka. Ovakvi programski jezici su, na primer, Pascal, Fortran, C, Algol, PL/1, itd. U ovakvim jezicima saopštava se ne samo **šta** program treba da uradi, već i **kako** to treba da uradi, tj. kojim redosledom izvršavanja koraka.

Druga važna klasa mašinski nezavisnih jezika su *deklarativni (neproceduralni)* programski jezici. Programima napisanim u ovim jezicima saopštava se računaru

šta treba da uradi, ali ne i kako, tj. ne saopštavaju se specifični koraci izvršenja, ili njihov redosled (tada je **kako** komponenta sadržana u jezičkom procesoru). U deklarativne paradigme spadaju, na primer, relacijska, funkcionalna, problemski-orijentisana, itd, sa jezicima kao što su Prolog, Lisp, Haskel, SQL, itd.

Posebno prihvćena je objektno-orijentisana paradigma, s obzirom da, kroz koncept objekata i njihovih odnosa, pruža dosta prirodnu podršku mentalnom modelu čoveka koji formuliše problem i modelira rešenje.

Relacijski programski jezici – logičko programiranje. Važna klasa neproceduralnih programskih jezika su logički programski jezici zasnovani na relacijama među podacima tj. na predikatima (logičkim izrazima).

Dve osnovne karakteristike logičkog programiranja su

1. upotreba činjenica i pravila za predstavljanje informacija i
2. upotreba dedukcije (izvođenja) za odgovaranje na pitanja

Korisnik (programer) formuliše činjenice i pravila, dok je sâm jezik za logičko programiranje (odnosno procesor tog jezika) zadužen za primenu dedukcije, tj. za kontrolu upravljanja.

Glavni predstavnik relacijskih – logičkih programskih jezika je programski jezik PROLOG (PROgrammation en LOGique – programiranje u logici) – jezik za programiranje u logici. Ovaj jezik je razvijen 1972. godine (Alain Colmarauer, Phillipe Roussel).

Činjenice ("podaci") programa na PROLOG-u predstavljene su konstrukcijama (predikatima) oblika

$P(X_1, \ldots, X_n)$

gde su $X_1, \ldots, X_n$ konstante ili promenljive. Značenje ove činjenice je da su $X_1, \ldots, X_n$ u relaciji P. Na primer, činjenica

otac(Marko, Ana)

ima značenje da su Marko i Ana u relaciji "otac", tj. da je Marko Anin otac.

Pravila (komponente programa, kao što je iskaz komponenta programa na proceduralnom jeziku) su oblika

$F(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \implies P(X_1, \ldots, X_n)$, što se zapisuje kao

$P(X_1, \ldots, X_n)$:– $F(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$.

$F(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ je logička formula, a $P(X_1, \ldots, X_n)$ je kao i u činjenici. Značenje pravila je da je predikat $P(X_1, \ldots, X_n)$ tačan ako je formula $F(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ tačna.

Na primer, pravilo

potomak(X,Y) :– otac(Y,X)

ima značenje da su X i Y u relaciji potomak ako (ali ne i samo ako) su Y i X u relaciji otac. Dakle, ovo pravilo kaže da ako je Y otac od X, onda je X potomak od Y.

Primer prološkog programa:

otac(Marko, Ana).
majka(Ana,Petar).
potomak(X,Y) :– otac(Y,X).
potomak(X,Y) :– majka(Y,X).
potomak(X,Y) :– otac(Y,Z), potomak(X,Z).

Poslednje pravilo ima sledeću interpretaciju:
Ako je Y otac od Z i X potomak od Z, onda je X potomak i od Y.

Izvršavanje prološkog programa zahteva se zadavanjem pitanja na koje program treba da proizvede odgovor.

Na primer, ako se postavi pitanje
?–otac(Marko,Ana)
odgovor će biti "DA" (u odgovarajućoj notaciji).

Ako se postavi pitanje
?–otac(Marko,Petar)
odgovor će biti "NE".

Ako se postavi pitanje
?–otac(Marko,X)
odgovor će biti "Ana".

Opšti oblik pitanja je, dakle, predikat
?–P($a_1, \ldots, a_n$)
gde su $a_1, \ldots, a_n$ – konstante ili promenljive, a odgovor na pitanje dobija se mehanizmom *unifikacije*.

Unifikacijom se izjednačavaju vrednosti promenljivih odnosno konstanti na istoj poziciji istog predikata (npr. pitanje ?–otac(Marko,X) daje odgovor "Ana" zato što postoji predikat otac(Marko,Ana); dakle, X se "unifikuje" sa "Ana").

Funkcionalni programski jezici – funkcionalno programiranje. Funkcionalno programiranje se karakteriše sledećim bitnim svojstvima:

1. Vrednost izraza zavisi samo od vrednosti podizraza, tj. nema bočnog efekta. Tako će, na primer uvek važiti da je $f(a) + f(b) = f(b) + f(a)$, što u C-u ili Pascal-u ne mora da važi. Ovo svojstvo funkcionalnih programskih jezika je posledica činjenice da u njima nema eksplicitnog iskaza dodele.

2. Rad sa memorijom je implicitan (nema eksplicitne deklaracije promenljivih kojima se dodeljuje memorijski prostor).

3. Funkcije se ponašaju kao elementarne vrednosti, tj. kao objekti prvog reda.

4. Jezik je netipiziran, tj. nema eksplicitnih tipova, pa se provera slaganja tipova može vršiti tek u vreme izvršavanja programa.

U funkcionalnom programskom jeziku ceo program je jedna funkcija. Funkcija se predstavlja listom – sekvencom od 0 ili više vrednosti – brojeva, karaktera, drugih

lista, drugih funkcija. Lista se zapisuje u paru malih zagrada. Prazna lista označava se sa ().

Prvi predstavnik funkcionalnog programiranja je programski jezik LISP (LISt Processing – obrada lista), razvijen još daleke 1958. godine (McCarthy). Do danas je razvijen veći broj dijalekata LISP-a i drugih funkcionalnih jezika – ML, SCHEME. Iswim, Miranda, itd.

Primer programa za nalaženje najvećeg zajedničkog delioca brojeva n, m, u stilizovanoj LISP-notaciji:

```
(nzd n m
     (if (eq n m)
          n
          (if (leq n m)
               (nzd (sub m n) n)
               (nzd (sub n m) m)
          )
     )
)
```

Ceo program je funkcija za računanje nzd(n,m). U programu se koriste i funkcije eq sa dva argumenta (za ispitivanje jednakosti argumenata), leq sa dva argumenta (za ispitivanje relacije <= među argumentima), sub sa dva argumenta (za oduzimanje drugog argumenta od prvog), if sa tri argumenta – liste, od kojih je prvi – funkcija kojom se testira uslov, drugi – funkcija koja se izračunava u slučaju da je uslov tačan, i treći – funkcija koja se izračunava u slučaju da uslov nije tačan. Funkcija nzd koristi i rekurzivne pozive iste funkcije (nzd).

Problemski-orijentisani jezici. Problemski orijentisani ili specijalizovani jezici namenjeni su rešavanju problema u posebnim oblastima primene. Osnovne klase podataka i operacija u takvim jezicima po pravilu su elementarne samo u problematici za koju su jezici namenjeni. Realizacija takvih podataka i operacija u programskim jezicima visokog nivoa opšte namene obično je daleko od jednostavne. Postojanje problemski orijentisanih jezika za neku oblast omogućuje stručnjaku iz te oblasti da pri korišćenju računara razmišlja na način koji je svojstven za oblast primene. Time se znatno povećava efikasnost rada, a od korisnika se traži samo elementarno poznavanje računara koji koristi. Ovi jezici su po pravilu proceduralni.

Primera problemski orijentisanih jezika ima mnogo. Na primer, jezici za rad sa bazama podataka – npr. SQL, jezici za obeležavanje teksta – npr. SGML, HTML, itd.

U jeziku SQL, na primer, iskaz DELETE FROM R WHERE IME = "M.Tomić" ima značenje "izbrisati iz tabele R slog(ove) koji u polju IME ima vrednost "M.Tomić" (tabela je smeštena na spoljašnjem mediju, npr. disku, a predstavljena je jednom ili većim brojem datoteka).

Ovaj primer pokazuju da se u problemski orijentisanim jezicima komandama jednostavne sintakse zadaju veoma složene radnje.

Objektno-orijentisani jezici. Objektna paradigma nastala je i razvijala se poslednje tri decenije u raznim oblastima računarstva – programskim jezicima, bazama podataka i sistemima za reprezentaciju znanja.

Objektno orijentisani programski jezici javili su se kao odgovor na softversku krizu, kao način da se, lokalizovanjem programskih segmenata uz klase podataka na koje se odnose, a ne uz aplikacije, dobije programski kôd koji se može ponovo upotrebiti – u raznim aplikacijama.

Objektno orijentisane (ili samo objektne) jezike karakteriše bogat skup vrsta podataka i mogućnost proširenja tog skupa korisničkim podacima.

Već smo se upoznali sa tipovima podataka (npr. u C-u) kao skupovima vrednosti i skupovima operacija nad tim vrednostima. U sličnom smislu, u objektnim jezicima se mogu definisati nove vrste podataka, ali mnogo bogatije – bez ograničenja na strukturu, kao i operacije koje se mogu primeniti na te vrste podataka. Ali, objektna paradigma uključuje i više od toga. Ona omogućuje definisanje vrste podataka prema obrascu neke već definisane vrste, sa dodatkom nekih "profinjenja" – novih komponenti i operacija, uz nasleđivanje svih komponenti i operacija polazne vrste. To je relacija nadtip/podtip (ili u objektnom modelu – nadklasa/podklasa).

Dakle, objektnu paradigmu pa i objektne programske jezike karakteriše niz koncepata i svojstava.

1. Objektni jezici polaze od podatka, kao dominantnog koncepta, i uz podatak vezuju operacije – to je *objekat*; objekat može biti vrednost 5 sa operacijama koje se mogu primeniti na tu vrednost, npr. dodaj(parametar), oduzmi(parametar), pomnoži(parametar), podeli(parametar), ali objekat može biti i složena struktura – npr. OSOBA, pa i ceo programski sistem sa primenljivim skupom operacija.

2. *Klasa* je skup objekata sa istim svojstvima (tzv. "instancnim promenljivim" – mogu se zamisliti kao polja neke složene strukture), i ponašanjem – operacijama. Na primer, INTEGER može biti jedna klasa.

3. Svaki objekat je *primerak* neke klase.

4. Objekti između sebe komuniciraju *porukama*. To nisu poruke tipa "Zdravo, kako si?" već su to poruke kojima se od objekta koji ih prima zahteva da izvrši neku od operacija definisanih u klasi kojoj pripada. Objekat na primljenu poruku odgovara izvršavanjem *metode* – operacije za koju samo objekti iz te klase znaju kako se izvršava (samo je njima poznata *implementacija metode*).

5. Nova klasa može se definisati kao *podklasa* već definisane klase (svoje *nadklase*). Podklasa *nasleđuje* operacije i svojstva od svoje nadklase, ali može imati i svoje sopstvene. Na primer, za definisanu klasu OSOBA može se definisati podklasa STUDENT, koja ima sva svojstva klase OSOBA, ali i neka specifična.

6. U objektnom modelu primenjuje se *skrivanje informacija* – objektu se može pristupiti samo preko poruka tj. metoda koje su definisane za klasu kojoj

objekat pripada. Sama struktura objekta – njegova svojstva, tj. instancne promenljive, nisu vidljive drugim objektima.

7. Objektni model karakteriše *učaurenje podataka i operacija* u celinu – objekat, što omogućuje jednostavnu izmenu strukture objekta i klase i implementacije operacija bez efekta na aplikacije koje taj objekat koriste (sve dok su definicije operacija nepromenjene – sa istim nazivima i istim parametrima).

8. U objektnim jezicima prisutno je preopterećenje operatora (svojstvo da se istom operatoru mogu pridružiti različita značenja; elementi preopterećenja operatora postoje i u Pascal-u – na primer + označava i operaciju sabiranja i operaciju unije, u zavisnosti od tipa na koji se primenjuje. Tako se svaka metoda koja se nasledi od nadklase u podklasi može redefinisati i reimplementirati.

Neki od prvih primera objektno orijentisanih jezika su SMALLTALK, OBERON, EIFFEL. Današnji objektni jezici su, na primer, JAVA, objektna proširenja proceduralnih jezika, npr. C++, C#.NET, Object Pascal, Python, Ruby, itd.