1

Uvod

U ovom uvodnom poglavlju se prvo prezentiraju osnovna svojstva algoritama. Navode se načini njihovog stvaranja, kao i načini njihovog zapisivanja. Zatim se u drugom dijelu ovog poglavlja, na općenit način opisuju strukture podataka, tipične operacije za rad nad tim strukturama, te se daju klasifikacije struktura podataka po različitim kriterijumima. Na kraju poglavlja se algoritmi i strukture podataka razmatraju u kontekstu razvoja softvera, pri tome naglašavajući njihov značaj za razvoj i implementaciju različitih softverskih sistema.

1.1 O algoritmima

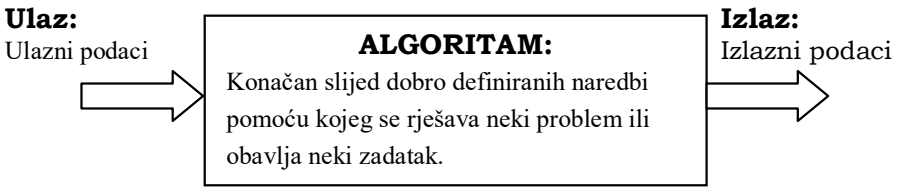
Pojam algoritam susrećemo u različitim situacijama u kojima se do određenog cilja dolazi izvođenjem operacija nekim zadanim redoslijedom, pri čemu je skup mogućih operacija obično prethodno precizno definiran. Smatra se da riječ algoritam dolazi od imena perzijskog matematičara *Muhamed ibn Musa al Horezmi*, koji je živio u devetom stoljeću. U svojoj knjizi „*Račun sa Hindu brojkama"*, on je opisao indijsku notaciju u kojoj vrijednost numerala zavisi od njegovog položaja. Ova notacija uključuje i nulu. Navedena knjiga zapravo opisuje pravila za obavljanje aritmetičkih operacija nad brojevima zapisanim u dekadskom sistemu. Knjiga je u 12. stoljeću prevedena na latinski jezik pod naslovom „*Algoritmi de numero indorum",* što bi u prevodu bio naslov *„Al Horezmi o indijskim brojevima"*. U ovom naslovu je upotrijebljena riječ *Algoritmi*, kao latinski oblik imena *Al Horezmi*. Nakon toga se postepeno riječ algoritam počela koristiti za pravila računanja sa indijskim brojevima, koji su postepeno zbog velikog utjecaja ove knjige nazvani arapskim brojevima. Original te knjige na arapskom jeziku je izgubljen, ali postoji prijevod na latinski. U početku su se pod pojmom algoritma podrazumijevala samo pravila za računanje brojevima. Danas se pod algoritmom podrazumijevaju pravila za obavljanje zadataka u različitim oblastima, a najčešće u računarstvu.

Pod algoritmom podrazumijevamo precizno definiranu proceduru u obliku uređenog skupa jednoznačnih izvedivih koraka. Uređeni skup koraka zapravo definira redoslijed, kojim će se ti koraci izvoditi. Algoritmi imaju sljedeća svojstva:

- obično imaju jedan ili više precizno definiranih ulaznih početnih objekata;
- kao rezultat izvršavanja imaju barem jedan izlazni objekat ili rezultat;
- moraju biti sastavljeni od konačnog broja koraka;
- moraju se završiti u konačnom vremenu;

- naredbe trebaju biti jasne i nedvosmislene, tako da se može izvesti svaki korak algoritma, te se naredbe moraju moći realizirati pomoću instrukcija računara u konačnom vremenu.

Slika 1.1 ilustrira značenje pojma algoritam.



Slika 1.1. Neformalna definicija algoritma

Bitno je istaknuti da se koraci nekog algoritma mogu izvoditi *sekvencijalno* ili *paralelno*. Kod paralelnih algoritama, koraci se mogu izvoditi istovremeno. Naime, paralelni algoritmi sadrže tzv. *niti* (eng. *threads*), koje izvodi pojedini procesor, što omogućuje da procesori obavljaju svoj dio zadatka.

Obavljanje algoritma se naziva **algoritamskim procesom**. Tijekom odvijanja algoritamskog procesa, postupno se izgrađuju izlazni objekti, pri čemu se u nekim fazama algoritamskog procesa mogu pojaviti i određeni međurezultati. Pod istim uvjetima algoritam se ponaša na predvidljiv i očekivan način, pa kažemo da algoritmi imaju **deterministično ponašanje**.

1.1.1 Zapisivanje i stvaranje algoritama

Općenito, algoritmi se mogu opisati npr. prirodnim govornim jezikom, pri čemu izvršitelj algoritma mora poznavati taj jezik. Isto tako, algoritmi se mogu opisati u grafičkoj formi u obliku dijagrama toka ili u nekom umjetnom jeziku kojeg obično nazivamo **programskim jezikom**. Algoritmi zapisani u nekom programskom jeziku se nazivaju **programi**. Za realizaciju nekog algoritma, neophodno ga je zapisati u jednom ili više stvarnih programskih jezika. Pisanje programa u programskim jezicima zahtijeva detaljno poznavanje zapisivanja instrukcija u tim jezicima. Međutim, važno je istaknuti da zapis algoritma u nekom programskom jeziku, zbog zahtjeva da se strogo koriste pravila dotičnog jezika, često zamagljuje osnovne zamisli koje su ugrađene u sam algoritam. Zbog toga se algoritmi često zapisuju i u obliku tzv. **pseudojezika**, jer to omogućuje njegov opis na način da se istaknu bitni elementi algoritma, pri tome zanemarujući neke detalje koji smanjuju njegovu razumljivost. S druge strane, algoritam zapisan u obliku pseudojezika se može u većini slučajeva na jednostavan način prevesti u konkretan programski jezik.

Algoritmi nastaju na različite načine. Jedan od izvora algoritama su znanstvene teorije. Na primjer, u matematici se često različite spoznaje zapisuju u obliku teorema, koje su zapravo izvor mnogih algoritama. Ovakve algoritme nazivamo **teorijskim algoritmima**. Jedan od bitnih izvora algoritama je i praktično iskustvo, koje uključuje opažanje pojedinih objekata i eksperimentiranje. Na temelju opažanja se zatim uspostavljaju različiti modeli,

koji na formaliziran način opisuju međuzavisnost pojedinih objekata, a ovisno o tipu, ti modeli mogu sadržavati i jasno definirane korake za različita izračunavanja, pa se takvi modeli lako pretvaraju u algoritme. Pošto ovi algoritmi nastaju tako da se promatraju različiti procesi u društvu ili prirodi, te budući da se eksperimentiranjem dolazi do spoznaja o mehanizmima djelovanja pojedinih objekata, algoritmi koji nastaju na taj način se nazivaju **eksperimentalnim algoritmima**.

Važan izvor algoritama je pristup koji se temelji na prethodno već kreiranim algoritmima. Naime, veliki broj **novih problema** koji se pojavljuju su problemi koji su u većoj ili manjoj mjeri **slični starim problemima** za koje su već stvoreni zadovoljavajući algoritmi, pa se stvaranje algoritama koji rješavaju nove probleme zapravo često može temeljiti na skupu već razvijenih algoritama, pomoću kojih su prethodno riješeni slični stari problemi. Pošto ovakav pristup uključuje metode kojima se konstruira novi algoritam na temelju prethodno definiranih algoritama, takvi algoritmi se često nazivaju **konstrukcijskim algoritmima**. Isto tako, jedan od izvora algoritama može biti domišljatost i kreativnost stvaratelja algoritma, koji u trenutku inspiracije rješavanjem nekog problema stvara novi algoritam.

Naravno, najveći broj algoritama je nastao nakon pojave računara, mada treba istaknuti da neki algoritmi koriste metode koje su razvijene i prije pojave računara. Na kraju, spomenimo da algoritmi mogu nastajati i kombinacijom različitih, prethodno navedenih pristupa.

1.1.2 Konvencije pseudojezika

Zbog nastojanja da se u prvi plan stave pristupi, ideje, kao i sama suština pojedinih algoritama, u ovoj knjizi je za njihov prikaz korišten pseudojezik. Korištenjem pseudojezika su određeni implementacijski detalji isključeni iz opisa algoritma, ali je zato postignut veći nivo razumljivosti najbitnijih elemenata prikazanih algoritama. Pseudojezik korišten u ovoj knjizi uvodi sljedeće konvencije:

- Pseudojezik sadrži kontrolne strukture **while**, **repeat** i **for**. Tijelo kontrolne strukture je uvučeno, dok je kraj označen pomoću konstrukcije **end + ime strukture**. Na primjer, kraj **while** petlje je označen kao **end_while**, dok je kraj **for** petlje označen kao **end_for**.
- Pseudojezik sadrži uobičajenu strukturu izbora **if**. Kraj ove strukture je također označen pomoću konstrukcije **end + ime strukture**, tj. pomoću oznake **end_if**.
- Postoji uobičajena dodjela vrijednosti, koja je označena znakom =. Na primjer, operaciju kojom se varijabli a dodjeljuje vrijednost varijable b, zapisujemo kao $a = b$. Osim toga, dozvoljena je višestruka dodjela vrijednosti sljedećeg oblika $a = b = c$, kojom se varijablama a i b dodjeljuje vrijednost izraza c. Drugim riječima, prethodno navedena višestruka dodjela se treba interpretirati kao ekvivalentna dodjeli $b = c$, nakon koje slijedi dodjela $b = a$.
- Za testiranje ekvivalentnosti dviju vrijednosti koristimo simbol ==.

- Postoji naredba razmjene vrijednosti dviju varijabli, koja koristi znak ↔. Na primjer, naredbom $a \leftrightarrow b$ označavamo razmjenu vrijednosti varijabli a i b. Drugim riječima, naredba $a \leftrightarrow b$ ima značenje sljedeće sekvence naredbi: $priv = a$, $a = b$, $b = priv$.
- Pristup elementu nekog niza ostvarujemo pomoću indeksa u uglatim zagradama iza imena niza. Na primjer, elementu niza *A*, koji se nalazi na poziciji s indeksom 5, pristupamo kao *A*[5]. Simbol „..“ koristimo da označimo neki raspon elemenata niza. Na primjer, oznaka *A*[2..6] označava sljedeće elemente: *A*[2], *A*[3], *A*[4], *A*[5] i *A*[6].
- Složeni podaci su organizirani u objekte, koji su sastavljeni od atributa. Nekom atributu pristupamo navođenjem imena objekta i tačke, nakon čega slijedi ime atributa. Na primjer, nizove tretiramo kao objekte sa atributom *duzina*, koji pokazuje koliko elemenata pojedini niz sadrži. Za označavanje broja npr. elemenata u nizu *A*, koristimo oznaku $A.duzina$.
- Parametri se prenose u procedure i funkcije po vrijednosti i po referenci. Mehanizam prijenosa sintaksno nije posebno označen. Kod funkcija se vrijednost vraća izračunavanjem izraza u naredbi **return**.
- Pri opisu pojedinih algoritama ili njegovih dijelova, ponekada se koriste i jezičke konstrukcije. Takvi opisi ne uključuju sve potrebne specifične detalje, pa ih je pri realizaciji algoritama u nekom od programskih jezika potrebno upotpuniti preciznijim i formalnijim strukturama iz konkretnog programskog jezika.
- Rekurzija je dozvoljena.
- Nekom polju zapisa na kojeg pokazuje pokazivač pristupamo navođenjem imena polja, a zatim imena pokazivača u malim zagradama. Na primjer, ako neki zapis, na koji pokazuje pokazivač *p*, sadrži polja *a*, *b* i *c*, onda pojedinim poljima pristupamo navođenjem $p.a$, $p.b$ i $p.c$, respektivno.
- Vrednovanje logičkih izraza koji uključuju operatore „and“ i „or“ je optimizirano. Drugim riječima, kada se vrednuje npr. izraz „*x* and *y*“, prvo će se vrednovati *x*. Ako se *x* vrednuje kao *false,* onda će se cjelokupni izraz vrednovati kao *false*, pa se izraz *y* u tom slučaju uopće ne treba vrednovati. S druge strane, ako se izraz *x* vrednuje kao *true*, tada je potrebno vrednovati i *y*, da bi se vrednovao cjelokupni izraz „*x* and *y*“. Slična je situacija i sa vrednovanjem izraza oblika „*x* or *y*“. Tada se izraz *y* vrednuje samo ako je *x* vrednovan kao *false*.
- Pretpostavlja se postojanje prethodno definiranih funkcija ili procedura, kao što su: ERROR (signalizacija greške), PRINT (ispis sadržaja na standardnom izlazu), GETNODE (alokacija memorijskog prostora za jedan element u dinamičkoj strukturi), FREENODE (oslobađanje zauzetog memorijskog prostora), itd. Sve procedure i funkcije, uključujući i one koje su predefinirane, su označene velikim slovima.

- Budući da su algoritmi detaljno objašnjeni u popratnom tekstu, komenari nisu uključeni u pseudokod.

1.1.3 Napomene o primjerima u jeziku C++

Iako je namjena knjige da pruži pregled i opis standardnih algoritama i struktura podataka na konceptualnom nivou i za čitaoce koji ne poznaju programski jezik C++, ipak su za većinu u ovoj knjizi opisanih algoritama i struktura podataka dati i primjeri njihove realizacije, pri čemu je za tu svrhu odabran jezik C++. Potrebno je istaknuti da ti primjeri podrazumijevaju poznavanje jezika C++ na odgovarajućem nivou, jer se u tim primjerima ne nalaze opisi elemenata koji se odnose na sam jezik C++, kao i opisi pojedinih detalja koji proizlaze iz specifičnosti, kako programskog jezika C++, tako i objektno orijentirane metodologije. Na primjer, procedure i funkcije u pseudokodu su opisane proceduralno, dok su prikazane implementacije analognih metoda u jeziku C++ realizirane objektno orijentiranim pristupom. Pa tako, na primjer, neka funkcija u pseudokodu koja se poziva kao NEKA-FUNKCIJA (A, a_1, a_2, .., a_n), pri čemu je argument A npr. neka struktura podataka, pri njenoj implementaciji kao analogne metode neke klase T u jeziku C++, može imati poziv `A.NekaFunkcija (a1, a2,...an)`, pri čemu se argument A pri zapisu algoritma u pseudokodu tretira kao ulazni argument, dok se u objektno orijentiranoj realizaciji predstavlja kao instanca klase T. Isto tako, istaknimo da ponekada prikazani primjeri C++ implementacija u određenoj mjeri sadrže nešto veći sadržaj implementacijskih detalja, u odnosu na odgovarajuće procedure i funkcije prikazane u pseudokodu, što je naravno najčešće rezultat nemogućnosti da se u svim slučajevima u pseudokodu opišu specifičnosti koje su relevantne za neki konkretni programski jezik, u kome će se algoritmi i strukture podataka realizirati. Nadalje, nazivi pojedinih metoda, kojima su implementirane opisane operacije nad strukturama podataka, najčešće su imenovane na način da npr. funkcija u pseudokodu sa imenom NEKA-FUNKCIJA, u C++ implementaciji ima ime `NekaFunkcija`. Međutim, treba naglasiti da to nije striktno ispoštovano, zbog različitih razloga. Uz prikazane primjere C++ realizacija se u pravilu nalaze i testni programi, koji omogućuju ispitivanje implementiranih funkcionalnosti. Iako ti testni primjeri zapravo nisu relevantni za prikaz samih C++ implementacija, oni su ipak stavljeni uz primjere, kako bi se pružila mogućnost ilustracije načina rada pojedinih algoritama, kao i funkcionalnosti implementiranih struktura podataka. Programski kod u primjerima je u pravilu organiziran u više datoteka, pri čemu se često u primjerima, koji dolaze kasnije u knjizi, koriste datoteke (programski kod) prikazane u ranijim primjerima. Prikazani primjeri imaju prije svega za cilj da ilustriraju moguće realizacije, pri čemu treba naglasiti da se te realizacije naravno mogu napraviti i na drugačije načine. Spomenimo, na primjer, da postoji jako dobra generička biblioteka STL (*Standard Template Library*), koja jeziku C++ daje visok nivo generalizacije i apstrakcije, kada su u pitanju algoritmi i strukture podataka. Ova biblioteka standardizira uobičajene komponente, kojima se mogu na jako elegantan način realizirati mnogi standardni algoritmi i strukture podataka.

1.2 O strukturama podataka

Pri razvoju programskih sistema, jedna od najvažijih odluka se odnosi na prikladan izbor struktura podataka. Tip podataka definira skup vrijednosti, koje neki entitet (varijabla, vraćena vrijednost funkcije, itd.) može imati. Definiranjem tipa podataka zapravo se definira i način interpretacije sadržaja memorije. Elementarni tipovi podataka su oni tipovi čiji objekti se ne mogu dijeliti na prostije dijelove. Elementarni tipovi se definiraju u okviru nekog programskog jezika (cjelobrojni, realni, logički, znakovni, itd.). Definiranjem elementarnih tipova se postiže određeni nivo apstrakcije, koji omogućuje da se sa tipovima podataka manipulira na logičkom nivou, te na taj način nije potrebno poznavanje prikaza podataka na mašinskom nivou.

Apstraktnim tipovima podataka (eng. *Abstract Data Type*) se dodatno povećava nivo apstrakcije prikaza i obrade podataka. Apstraktnim tipom podataka se definira model tog tipa, koji uključuje definicije vrijednosti i definicije operacija. Da bi se implementirali apstraktni tipovi podataka, potrebno je specifikacije sa logičkog nivoa prevesti na neki konkretni programski jezik, na način da se apstraktni tipovi grade od elementarnih tipova podataka, koji su definirani u konkretnom programskom jeziku. Kao rezultat takve izgradnje nastaju organizirani skupovi podataka, koji uključuju elemente istog ili različitog tipa, koji su međusobno povezani na različite načine. Tako organizirani skupovi elemenata se nazivaju **strukture podataka**. Važno svojstvo struktura podataka je i **skup operacija**, koje se mogu izvoditi nad podacima koji su predstavljeni dotičnom strukturom.

U programskim jezicima se koriste dva osnovna načina za objedinjavanje logički povezanih elemenata: **nizovi** (eng. *arrays*) i **zapisi** (eng. *records*). Nizovi su homogene strukture koje sadrže elemente istog tipa, za razliku od zapisa koji je nehomogena struktura i može sadržavati elemente različitog tipa. Objedinjavanjem više elemenata u niz ili zapis, nastaje struktura koja omogućuje obradu logički povezanih podataka kao cjeline. Te strukture se mogu dalje kombinirati sa objektima istog ili različitog tipa, s ciljem stvaranja struktura proizvoljne složenosti.

1.2.1 Klasifikacija struktura podataka

Strukture podataka se mogu klasificirati na različite načine. Jedan od mogućih kriterija se temelji na međusobnim relacijama elemenata u strukturi podataka. Ako je neki element strukture samo u relaciji sa svojim prethodnikom i sljedbenikom, onda takvu strukturu nazivamo **linearnom strukturom**. Primjeri linearnih struktura su: niz, povezana lista, stek, red, itd. S druge strane, ako elementi strukture mogu biti u relaciji sa više drugih elemenata, onda takvu strukturu nazivamo **nelinearna struktura**. Primjeri nelinearnih struktura su stablo i graf.

Sljedeći kriterij koji možemo koristiti za klasifikaciju struktura podataka je mogućnost promjene njene veličine, pri izvršavanju programa. S obzirom na ovaj kriterij, strukture podataka se dijele na **statičke** i **dinamičke**. Kod statičkih struktura podataka, veličina je nepromjenjiva i definira se pri prevođenju. Prema tome, veličinu strukture podataka je potrebno definirati uzimajući u obzir maksimalan mogući broj elemenata, koji se može pojaviti pri izvršavanju programa, pa upotreba statičkih struktura ne omogućuje optimalno korištenje memorijskih resursa. S druge strane, dinamičke strukture koriste mehanizme dinamičke alokacije i dealokacije memorije, pa te strukture imaju svojstvo da

im se veličina može mijenjati za vrijeme izvršavanja programa, sukladno trenutnim potrebama. Dinamičke strukture podataka omogućuju efikasnije korištenje memorijskih resursa, jer one zauzimaju samo onoliko prostora, koliko je potrebno u nekoj stvarnoj primjeni.

Strukture podataka koje se pohranjuju na vanjskim memorijama se nazivaju **datoteke**. Obično se kao osnovni element datoteke uzimaju **zapisi**. Podaci u datotekama mogu biti organizirani na različite načine, što prije svega uključuje način fizičkog poretka pohranjenih zapisa. Osim toga, način organizacije datoteke je definiran i skupom operacija pomoću kojih se pristupa pojedinim zapisima.

1.2.2 Memorijski prikaz struktura podataka

Nakon definiranja logičke organizacije strukture podataka specificiranjem sastavnih dijelova i veza između njih, potrebno je definirati i način prikaza strukture podataka u memoriji, što zapravo predstavlja način njene fizičke implementacije. Načine memorijskog prikaza struktura podataka možemo klasificirati na:

- sekvencijalne (kontinualne) prikaze;
- ulančane (nekontinualne) prikaze.

Kod **sekvencijalnog** prikaza, elementi strukture su raspoređeni u kontinualnom memorijskom prostoru. Elementi mogu biti istog ili različitog tipa. Za pristup elementima je potrebno poznavati početnu adresu, indeks pozicije i veličinu pojedinih elemenata. Pošto sam fizički raspored elemenata istovremeno možemo iskoristiti i za prikaz odnosa elemenata na logičkom nivou, onda se sekvencijalne strukture često koriste za memorijski prikaz linearnih struktura. Isto tako, sekvencijalne strukture se mogu koristiti i za prikaz nelinearnih struktura.

Kod ulančanog prikaza, elementi strukture su raspoređeni na proizvoljnim mjestima u memorijskom prostoru, koja u općem slučaju ne tvore kontinualan memorijski prostor. Prema tome, kod ulančanog prikaza, fizički poredak ne odgovara logičkom poretku elemenata. Da bi se ipak mogao izraziti logički poredak, koriste se **pokazivači** kao poseban elementarni tip podataka. Pokazivači kao skup vrijednosti imaju memorijske adrese nekih drugih objekata u memoriji. Svaki element strukture, pored informacionog sadržaja, ima jedan ili više pokazivača, koji sadrže adrese nekih drugih elemenata strukture. Pošto se ulančanim prikazom mogu izraziti odnosi sa više drugih elemenata strukture, općenito možemo reći da je on prikladan za realizaciju nelinearnih struktura. Međutim, ulančani prikazi se mogu koristiti i za realizaciju linearnih struktura, ako se te strukture žele implementirati kao dinamičke strukture, ili ako se ulančanim prikazom dobivaju bolje performanse za obavljanje određenih operacija, u odnosu na sekvencijalni prikaz. Jedan od ključnih nedostataka ulančanog prikaza ogleda se u sporijem pristupu elementima strukture, jer je za pristup željenom elementu potrebno slijediti niz pokazivača. Suprotno tome, pristup elementima kod sekvencijalnog prikaza je direktan, što u mnogim situacijama omogućuje dizajniranje struktura podataka nad kojima će se moći efikasnije obavljati određene operacije. Osim toga, ulančani prikaz zahtijeva dodatni memorijski prostor za smještanje pokazivača, pa u nekim situacijama treba razmotriti i učinak na ukupne performanse. Međutim, općenito govoreći, ulančani prikaz omogućuje efikasnije korištenje memorijskih resursa, zbog korištenja dinamičke alokacije i dealokacije memorije.

Prema tome, oba memorijska prikaza zapravo imaju svoje prednosti i mane, pa je za prikladan izbor potrebno u svakoj konkretnoj situaciji analizirati i razmotriti zahtjeve, koji se odnose na željene karakteristike same strukture. Pri takvim analizama je, između ostalog, potrebno voditi računa o frekvenciji obavljanja pojedinih operacija, te implementaciju strukture temeljiti na prikazu koji garantira veću efikasnost za frekventnije operacije. Nadalje, sekvencijalni i ulančani prikazi se mogu u određenim situacijama kombinirati, posebno kada se nastoje iskoristiti prednosti i jednog i drugog prikaza.

1.2.3 Operacije sa strukturama podataka

Prethodno je već rečeno da je jedan od ključnih faktora, koji treba uzeti u obzir pri implementaciji struktura podataka, skup operacija koje će se izvršavati nad elementima strukture. Neke od tipičnih operacija su: pristup elementima radi čitanja ili dodjele nove vrijednosti, obilazak svih elemenata, pretraživanje, umetanje novog elementa, brisanje elementa strukture, itd. Dakako, pojedine strukture podataka imaju i operacije koje su specifične samo za tu strukturu.

Operacija obilaska podrazumijeva da se na sistematičan način svim elementima strukture pristupi samo po jednom, što je vrlo jednostavno napraviti kod linearnih struktura. S druge strane, kod nelinearnih struktura je operacija obilaska nešto složenija, te postoji više različitih algoritama, koji istovremeno daju i različit poredak obilaska.

Operacijom pretraživanja se pronalazi element u strukturi sa traženim sadržajem. Ako je struktura neuređena po sadržaju elemenata, pretraživanje se svodi na obilazak i usporedbu elemenata strukture sa traženim sadržajem. Operacija se završava pronalaskom elementa sa traženim sadržajem (uspješno pretraživanje) ili obilaskom svih elemenata, što omogućuje da se utvrdi da element sa traženim sadržajem nije prisutan u strukturi (neuspješno pretraživanje). S druge strane, ako je struktura podataka uređena, onda se operacija pretraživanja može obaviti efikasnije primjenom odgovarajućih algoritama. Osim toga, kod uređenih struktura podataka nije potrebno obići sve elemente da bi se utvrdilo da neki traženi element nije prisutan u strukturi podataka.

Umetanje novog elementa je operacija na koju se obično postavlja zahtjev da pri njenom obavljanju ne dolazi do premještanja već prisutnih elemenata u strukturi podataka. Ovakav zahtjev se obično postiže primjenom ulančanog memorijskog prikaza, mada ponekada postoje i sekvencijalne implementacije, koje ispunjavaju dati zahtjev. Osim toga, ova operacija se može efikasnije izvršiti kada se novi elementi umeću u neuređenu strukturu podataka. S druge strane, kod umetanja u uređenu strukturu, zahtijeva se održavanje poretka elemenata po sadržaju, pa je potrebno pronaći odgovarajuće mjesto za novi element, što umetanje u uređenu strukturu čini neefikasnijom operacijom u odnosu na umetanje u uređenu strukturu.

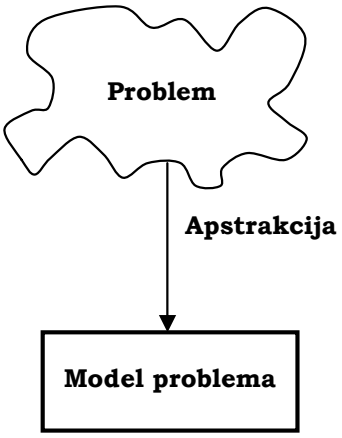
Brisanje elemenata je također operacija na koju se postavlja zahtjev da pri njenom obavljanju ne dolazi do pomjeranja ostalih elemenata prisutnih u strukturi. Ovaj zahtjev je lakše ispuniti kod ulančanog prikaza, jer se fizičkim uklanjanjem elementa koji se briše, ne narušava nekontinualni karakter memorijskog prikaza, a povezivanje razdvojenih dijelova strukture se svodi na preusmjeravanje pokazivača. S druge strane, kod sekvencijalnog prikaza, nakon fizičkog uklanjanja elementa koji se briše, narušava se kontinualni karakter memorijskog prikaza, pa se obično preostali elementi moraju premještati da bi se struktura

nakon brisanja reorganizirala na način da se popune upražnjene lokacije u strukturi, te na taj način obnovi njen kontinualni karakter.

1.3 Algoritmi i strukture podataka u kontekstu razvoja softvera

1.3.1 Apstraktni pogled na problem i kreiranje modela

U stvarnom svijetu i u različitim oblastima ljudske djelatnosti susrećemo se sa određenim specifičnim problemima koji se mogu riješiti stvaranjem programskih rješenja, što najčešće uključuje dizajniranje i implementaciju, za datu situaciju, prikladnih algoritama i struktura podataka. Međutim, problemi u stvarnom svijetu su često **nejasni** i **neodređeni**, pa je jedan od prvih koraka razumijevanje problema da bi se **odvojili bitni elementi od onih koji nisu bitni**. Drugim riječima, nastojimo stvoriti **svoj vlastiti apstraktni pogled na problem**, odnosno nastojimo kreirati **model problema**. Ovaj proces modeliranja se naziva **apstrakcija** i ilustriran je na slici 1.2.

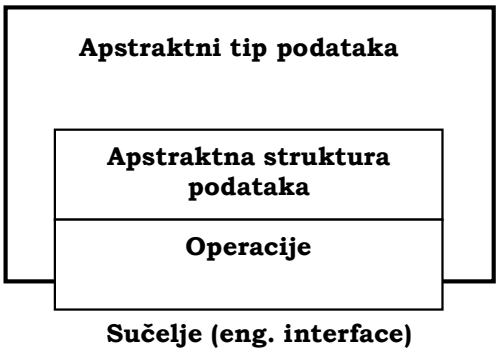


Slika 1.2. Kreiranje modela problema

Apstraktnim pogledom na problem, fokusiramo se samo na relevantnu „građu“ problema, da bismo identificirali bitna svojstva problema. Ta svojstva uključuju:

- podatke koji su relevantni za rješavanje datog problema;
- operacije pomoću kojih manipuliramo podacima.

Apstrakcija je strukturiranje nejasnih problema u dobro definirane entitete, da bi se mogli ispravno obrađivati. Ti entiteti definiraju strukturu podataka za neki skup elemenata. Strukturi podataka se može pristupati preko definiranih operacija. Entitet sa opisanim osobinama se naziva *apstraktni tip podataka.* Ovaj pojam je prethodno već uveden i opisan u odjeljku 1.2, dok je na ovom mjestu na slici 1.3 dodatno ilustrirano njegovo značenje.



Slika 1.3. Apstraktni tip podataka

Apstraktni tipovi podataka se mogu implementirati na različite načine, što prije svega ovisi o konkretnom programskom jeziku, u kojem će se implementirati neki programski sistem. Na primjer, u objektno orijentiranim jezicima, za prikaz apstraktnih tipova podataka se koriste *klase*.

1.3.2 Značaj algoritama i struktura podataka

Iz prethodnog opisa možemo ukratko reći da pod pojmom struktura podataka podrazumijevamo način organizacije podataka i informacija, dok pod terminom algoritam podrazumijevamo način manipulacije i obrade podataka. Algoritmi i strukture podataka predstavljaju ključne elemente za implementaciju različitih programskih rješenja. Da bismo mogli dizajnirati programe sa kvalitetno napravljenim programskim kodom, potrebno je da poznajemo neke temeljne i često korištene algoritme i strukture podataka. Drugim riječima, možemo reći da se pisanje kvalitetnog programskog koda sastoji od dizajniranja ili izbora prikladnih struktura podataka i od dizajniranja efikasnih algoritama, koji rješavaju dati problem. Pod kvalitetno napravljenim programskim kodom podrazumijevamo program koji *efikasno* rješava postavljeni problem, uzimajući u obzir i postavljena ograničenja (vrijeme, broj programera i projektanata, itd.). Prema tome, ne mogu se stvarati dobra programska rješenja bez poznavanja temeljnih i često korištenih algoritama i struktura podataka.

Kada spomenemo riječ programiranje, ono na što prvo pomislimo je pisanje programa u nekom od programskih jezika. U školama i na fakultetima se u pravilu pišu programi koji rješavaju **relativno jednostavne probleme**. Međutim, za dizajniranje i razvoj složenijih softverskih proizvoda, koji će svoju upotrebu naći u stvarnom svijetu, potrebno je koristiti mnoge tehnike dizajniranja, standarde kodiranja, metode testiranja itd., da bi se uopće razvio softver koji zadovoljava određene standarde kvalitete.

Postavlja se pitanje zašto u situacijama kada trebamo pristupiti razvoju nekog softverskog rješenja nije prikladno odmah sjesti za računar i početi pisati programe. Odnosno, da li je korištenje različitih metodologija zapravo 'gubljenje vremena'? Odgovor na prethodno pitanje je negativan. Naime, kako programi postaju sve veći i veći, potrebno je pažnju usmjeriti ne samo na kodiranje, nego i na neka druga pitanja vezana uz razvoj softvera. Potrebna znanja i vještine za razvoj softvera, stiču se u okviru oblasti koja se naziva **softverski inženjering**. Ova oblast kao predmet proučavanja ima pristup dizajnu, razvoju, proizvodnji i održavanju računarskih programa, pri čemu se za razvoj koriste razni

alati, koji inženjerima pomažu da upravljaju veličinom i kompleksnošću rezultirajućih softverskih proizvoda.

Ako je riječ o razvoju softvera koji uključuje na desetine hiljada linija programskog koda, te ako se razvoj odvija u timskom okruženju, onda su u **ukupni životni ciklus nekog softverskog proizvoda** uključene sljedeće aktivnosti:

- analiza problema;
- definicija zahtjeva;
- projektiranje na visokom i niskom nivou;
- implementacija;
- testiranje i verifikacija;
- isporuka softvera;
- upotreba softvera;
- održavanje softvera.

Mnoge od prethodno navedenih aktivnosti se izvode istovremeno. Na primjer, dok se kodira jedan od dijelova softverskog proizvoda, u isto vrijeme može se odvijati proces dizajniranja nekog drugog dijela tog istog softverskog proizvoda. Česta je situacija da određeni broj inženjera radi na jednom dijelu razvoja softvera, dok istovremeno druga grupa inženjera radi na nekom drugom dijelu. Upravljanje svim aktivnostima u takvim slučajevima nije jednostavan zadatak.

Ciljevi dobro napravljenog softvera su:

- Treba tačno i u potpunosti obaviti zadatak za koji je dizajniran.

  Prema tome, prvi korak u razvojnom procesu je tačno i precizno definiranje zahtjeva koje softver treba ispuniti. Kada je riječ o programima koje studenti dobivaju na vježbama i ispitima, ti zahtjevi su definirani od strane nastavnika ili asistenta, u vidu zadataka. S druge strane, kada je riječ o razvoju softvera za primjenu u stvarnom životu, tada dokument koji opisuje zahtjeve može sadržavati na stotine stranica.

- Kvalitetan softver treba biti takav da se može mijenjati.

  Promjene nekog softvera se događaju u svim fazama njegovog postojanja. Nije rijetkost da naručilac softvera na jedan način prvobitno specificira svoje zahtjeve koji opisuju željeni softver, a da nakon nekog vremena u manjoj ili većoj mjeri promjeni svoje zahtjeve. Dakako, promjene se dešavaju i u fazi kodiranja. Na primjer, možda smo upravo u toj fazi pronašli neko novo i bolje rješenje za postavljeni problem. Promjene se događaju i u fazi testiranja. Ako nam program daje pogrešne rezultate, morat ćemo napraviti neophodne ispravke.

U školskom okruženju razvoj programa završava kada riješimo postavljeni zadatak i dobijemo ocjenu. Međutim, u stvarnom svijetu promjene u softveru se događaju i u fazi održavanja. Česta situacija je otkrivanje onih grešaka u ovoj fazi, koje nisu otkrivene npr. u fazi testiranja. Isto tako, u postojeći softver često se žele dodati neke nove funkcije, mogućnosti obrade, itd. Prema tome, promjene u softveru su česte i događaju se u svim

fazama njegovog životnog ciklusa. Imajući to na umu, inženjeri koji rade na razvoju softvera trebaju pokušavati razvijati programe koji se mogu relativno lako modificirati. Nadalje, treba istaknuti činjenicu da se modifikacije u programu često izvode od strane programera koji uopće nije izvorni autor tog programa, nego je riječ o programeru koji radi na održavanju programa kojeg je neko drugi razvio.

Navedimo neke elemente koji doprinose da se programi mogu lakše mijenjati:

- Program bi trebao biti čitljiv, razumljiv i dobro dokumentiran.
- Velike programe treba podijeliti na manje logičke cjeline, koje su relativno nezavisne jedna od druge.
- Kvalitetan softver treba biti višekratno iskoristiv.

  Jedan od načina da se uštedi vrijeme za dizajniranje softverskih rješenja je ponovna upotreba programa, klasa, funkcija i drugih komponenti koje su razvijene u prethodnim projektima. Važno je istaknuti da je za kreiranje softvera, koji se može višekratno koristiti, potrebno uložiti dosta napora u fazama specifikacije i dizajniranja.
- Kvalitetan softver se treba završiti u predviđenim rokovima.

  U stvarnom svijetu je vrlo važno da se razvoj planiranog softvera završi na vrijeme, poštujući zadate rokove. Ovaj zahtjev je od velike važnosti za rad različititih institucija i organizacija u administrativnom, poslovnom i edukacijskom okruženju.

Na kraju ovog uvodnog dijela još jednom istaknimo da su algoritmi i strukture podataka osnovni elementi pomoću kojih se grade programski sistemi. Prema tome, poznavanje algoritama i struktura podataka je vrlo važno za implementaciju kvalitetnih programskih rješenja i kompleksnih softverskih sistema.

1.4 Pitanja i zadaci za vježbu

1. Ukratko objasnite pojam algoritma, te navedite neka ključna svojstva algoritama.
2. Navedite načine nastajanja algoritama i ukratko obrazložite specifičnosti svakog od tih načina.
3. Ukratko obrazložite šta podrazumijevamo pod determinističkim ponašanjem algoritama.
4. Ukratko obrazložite razliku između linearnih i nelinearnih struktura podataka.
5. Ukratko obrazložite razliku između statičkih i dinamičkih struktura podataka.
6. Ukratko obrazložite ključne razlike između sekvencijalnog i ulančanog memorijskog prikaza struktura podataka.
7. Navedite neke tipične operacije u radu sa strukturama podataka, te ukratko obrazložite efikasnost svake od tih operacija, uzimajući u obzir način na koji je struktura podataka memorijski prikazana (sekvencijalno ili ulančano), te uzimajući u obzir uređenost strukture podataka.
8. Ukratko obrazložite šta podrazumijevamo pod kreiranjem vlastitog apstraktnog pogleda na problem, odnosno pod kreiranjem modela problema.
9. Ukratko obrazložite šta podrazumijevamo pod apstraktnim tipom podataka.
10. Navedite aktivnosti (faze) koje su uključene u ukupni životni ciklus nekog softverskog proizvoda.
11. Navedite osnovna svojstva kvalitetnih programskih rješenja.
12. Koja svojstva trebaju imati programi da bi se mogli lakše mijenjati?
13. Ukratko obrazložite značaj algoritama i struktura podataka za softver inženjering, odnosno ukratko obrazložite zašto je važno poznavanje algoritama i struktura podataka za uspješan razvoj softverskih proizvoda.