Heš mapa (nastavak)

Heš mapa – operacije

Osnovne operacije:

- **dodaj** (insert) – dodavanje novog elementa u mapu
- **traži** (find) – traženje elementa u mapi sa zadatim ključem
- **obriši** (remove) – uklanja element sa zadatim ključem iz mape

Dodatne operacije:

- **pribaviDužinu** (getLength) – vrati veličinu mape
- **pribaviFaktorOpterecenja** (getLoadFactor) – vrati faktor popunjenosti

Implementacija **primarne transformacije** (heš funkcije) i eventualno **sekundarne transformacije**

Heš mapa – postupak dodavanja

Opšti postupak kod dodavanja podataka sa ključem *k*:

1. Izračunavanje heš funkcije h(k) na osnovu vrednosti ključa
2. Ako je pozicija sa izračunatim indeksom slobodna, podatak se upisuje na to mesto
3. U suprotnom, lokacija je zauzeta i došlo je do kolizije
    a) Vrši se rešavanje kolizije nekom od metoda
    b) Menja se vrednost primarne adrese u sekundarne adrese i pokušava upis sve dok se ne nađe slobodna lokacija
    c) Ako slobodna lokacija postoji, vrši se upis
    d) U suprotnom, mapa je puna

Rešavanje kolizija

Princip golubarnika (engl. *pigeonhole principle*) kaže da ne možemo izbeći sve kolizije

- Izračunavanje heša bez kolizija za *n* ključeva sa *m* slotova gde je *n* > *m*

Moguća rešenja:

- **Ulančavanje sinonima** ili **posebno ulančavanje** (engl. *separate chaining*), poznato i kao **otvoreno heširanje** (engl. *open hashing*) – kreira se mali rečnik za svaki ulaz
- **Otvoreno adresiranje** (engl. *open addressing*), poznato i kao **zatvoreno heširanje** (engl. *closed hashing*) – pokušati sa nekim sledećim mestom unutar heš mape

Ulančavanje sinonima

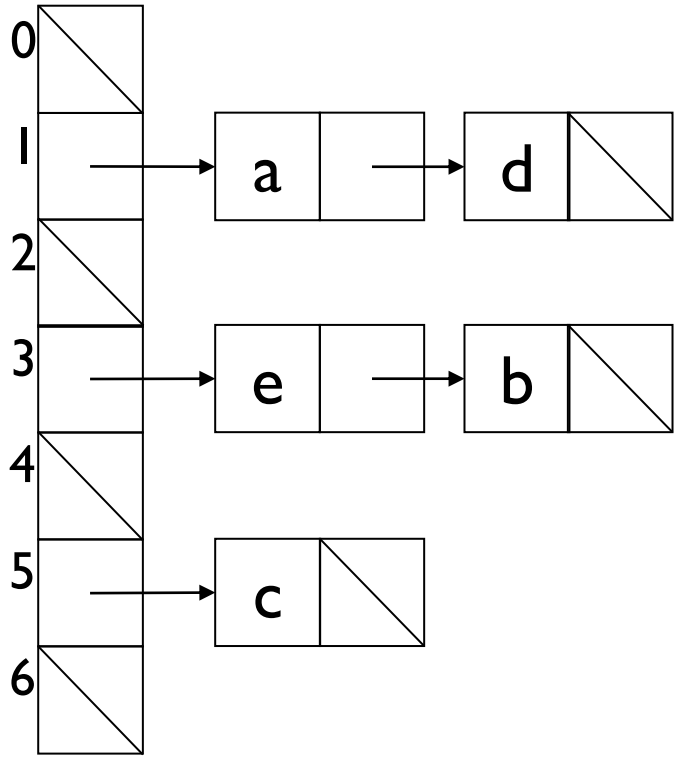
Kreira se mali "rečnik" za svako mesto

- Najčešće se koristi neuređena spregnuta lista, ali se može upotrebiti i binarno stablo traženja ili heš mapa

Svojstva:

- Faktor opterećenja λ može biti veći od 1
- Performanse rada sa heš mapom opadaju sa porastom dužine lanaca (u najgorem slučaju svode se na performanse spregnutih listi)
- Primena binarnih stabala traženja ili heš mapa umesto spregnutih listi može ubrzati sekundarno traženje

h(a) = h(d)
h(e) = h(b)



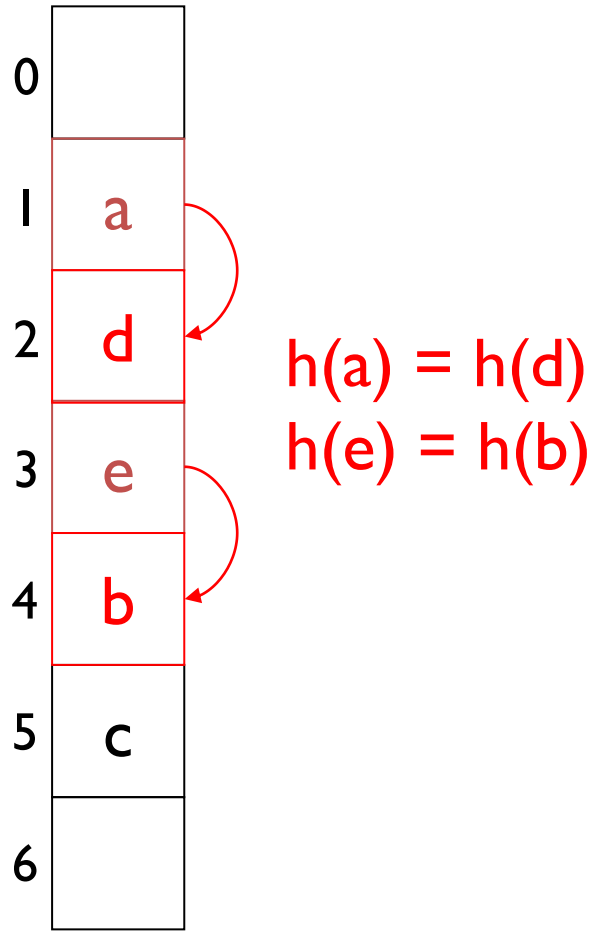
Otvoreno adresiranje

Na svako mesto u heš mapi može se preslikati samo jedan ključ

- Ako se dva ključa heširaju u istu adresu, ne mogu oba biti i smeštena na njoj
- Radi po principu "ko prvi stigne, prvi bude uslužen" (engl. *first-come, first-served – FCFS*)
- Sledeći ključ koji se heširа u istu adresu mora se preusmeriti na drugu adresu

Svojstva:

- Faktor opterećanja $\lambda \leq 1$
- Performanse opadaju sa "težinom" pronalaženja dobrog mesta za upis



Pretraga (engl. *probing*)

Neophodna kod **otvorenog adresiranja**, zahteva **funkciju za rešavanje kolizija** *f*(*i*) (engl. *collision resolution function*) tj. **sekundarnu transformaciju**

Pretraga:

- Prvi pokušaj – ako je dat ključ *k*, hešira se kao *h*(*k*)
- Drugi pokušaj – ako je *h*(*k*) zauzeto, probati *h*(*k*) + *f*(1)
- Treći pokušaj – ako je *h*(*k*) + *f*(1) zauzeto, probati *h*(*k*) + *f*(2)
- ...

Svojstva pretrage:

- Postavlja se *f*(0) = 0
- *i*-ti pokušaj je (*h*(*k*) + *f*(*i*)) mod *veličina*
- Ako *i* dođe do *veličina*-1, pretraga je neuspešna
- Duge sekvence pretraga dovode do loših performansi

Linearna pretraga

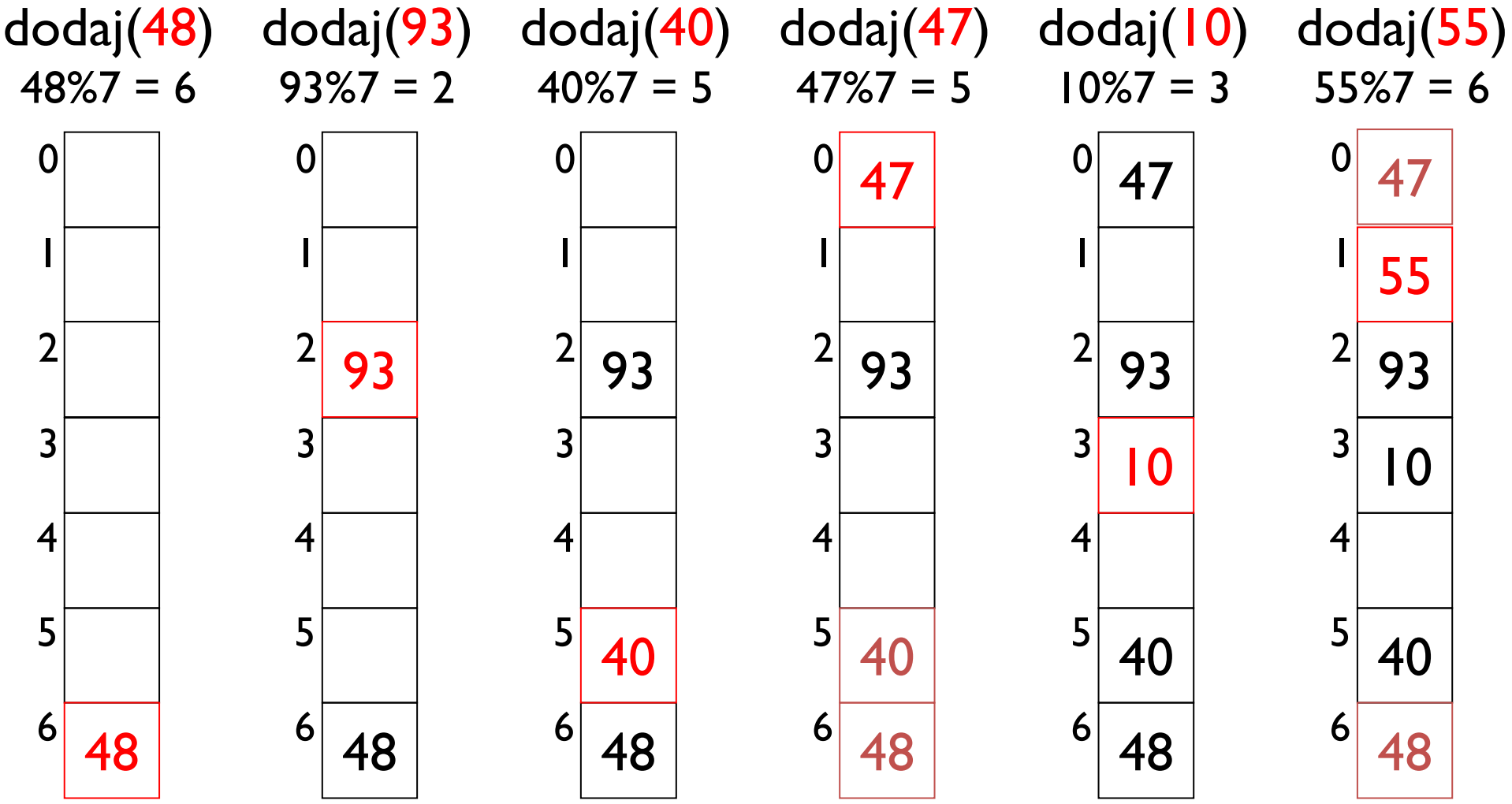
Funkcija sekundarne transformacije je:

$$f(i) = i$$

Sekvenca pretrage je:

- $h(k)$ mod *veličina*
- $(h(k) + 1)$ mod *veličina*
- $(h(k) + 2)$ mod *veličina*
- $(h(k) + 3)$ mod *veličina*
- ...

Linearna pretraga – primer



Kvadratna pretraga

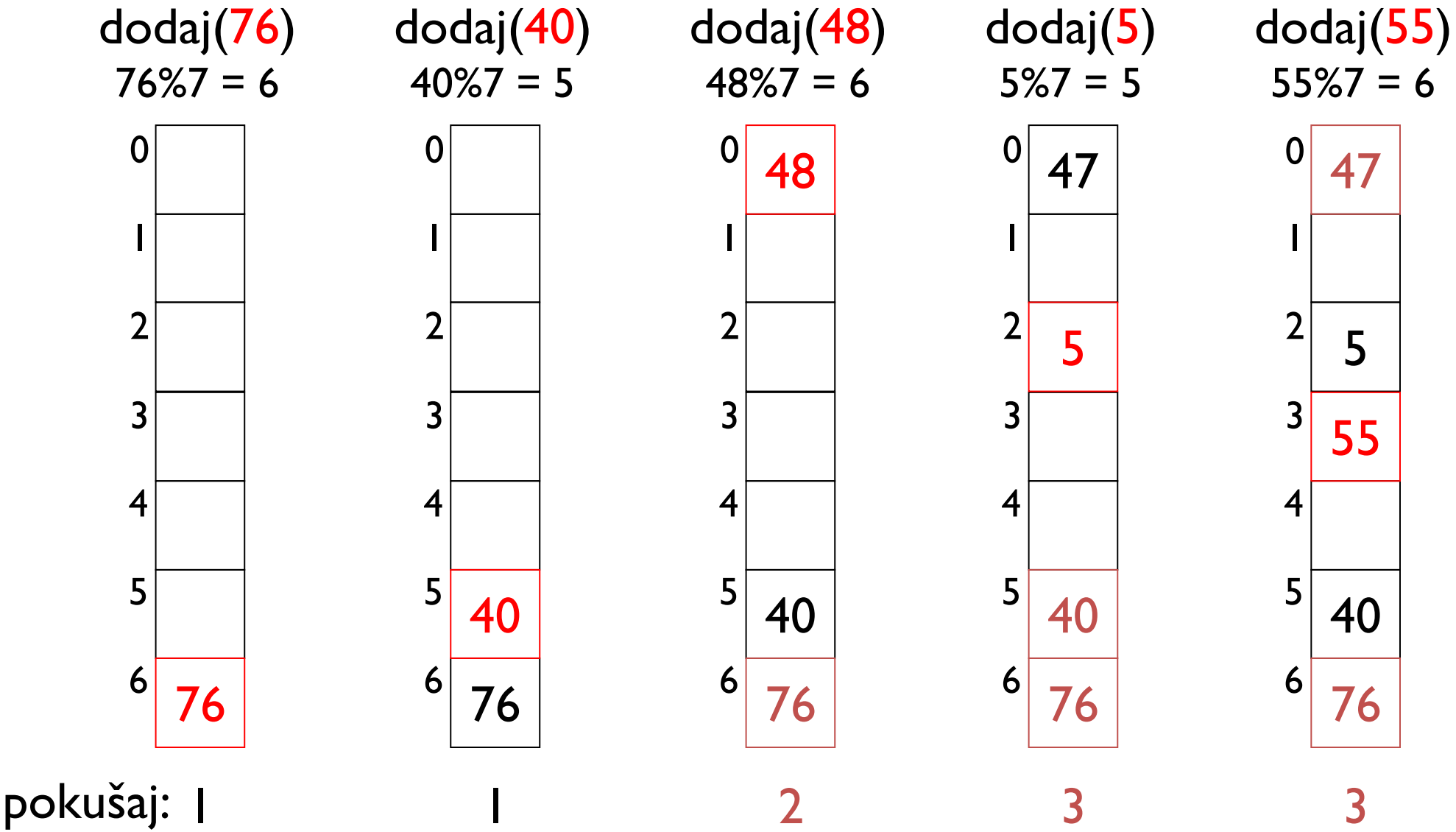
Funkcija sekundarne transformacije je:

$$f(i) = i^2$$

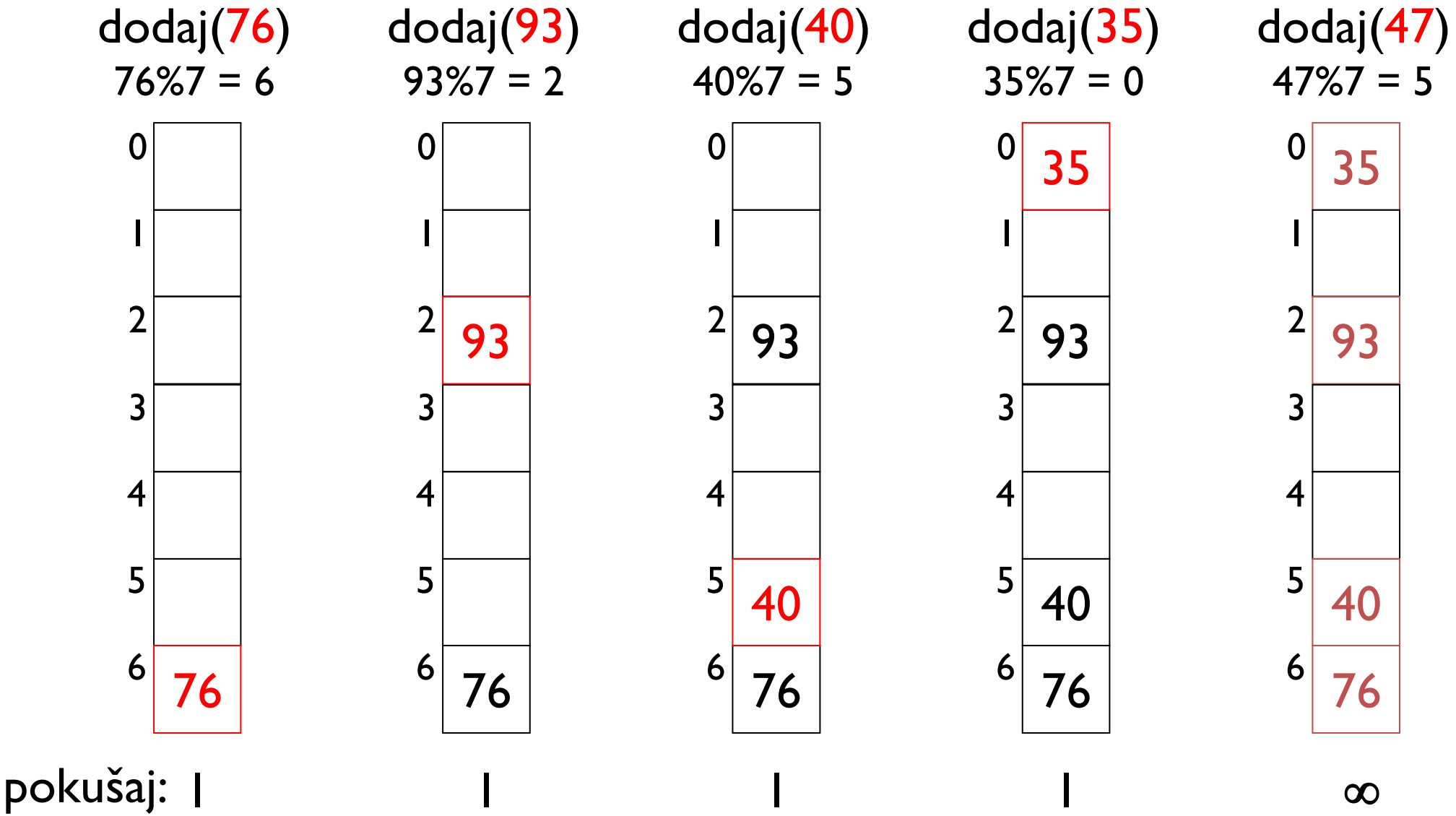
Sekvenca pretrage je:

- $h(k)$ mod *veličina*
- $(h(k) + 1)$ mod *veličina*
- $(h(k) + 4)$ mod *veličina*
- $(h(k) + 9)$ mod *veličina*
- …

Kvadratna pretraga – dobar primer



Kvadratna pretraga – loš primer



Dvostruko heširanje

Funkcija sekundarne transformacije je:

$$f(i) = i * heš_2(k)$$

Sekvenca pretrage je:

- $h_1(k)$ mod *veličina*
- $(h_1(k) + 1 \cdot h_2(x))$ mod *veličina*
- $(h_1(k) + 2 \cdot h_2(x))$ mod *veličina*
- …

Reheširanje

Kada **faktor opterećenja** λ **postane preveliki** (iznad određenog konstantnog praga), **reheširati sve elemente u novu, veću mapu**

- Traži $O(n)$, ali se amortizuje na $O(1)$ dok god otprilike dvostruko povećavamo veličinu mape prilikom realokacije
- Može drastično da unapredi performanse kroz poboljšano "raspršivanje" ključeva
- Sprečava neuspešnu pretragu kod tehnike otvorenog adresiranja
- Dozvoljava da se krene od male mape, a da se potom kreira proizvoljno velika mapa u skladu sa potrebama

Perfektno heširanje

Perfektno heširanje se koristi kada je **unapred poznat ceo skup mogućih ključeva**, npr. rezervisane reči u programskim jezicima, spisak fajlova na DVD disku, rečnik, itd.

Tada nam **perfektno heširanje** (engl. *perfect hashing*) dozovljava da u najgorem slučaju postignemo:

- Vremensku složenost $O(1)$
- Prostornu složenost $O(n)$

Perfektno heširanje

Statički skup n poznatih ključeva

Ulančavanje sinonima, heširanje na dva nivoa

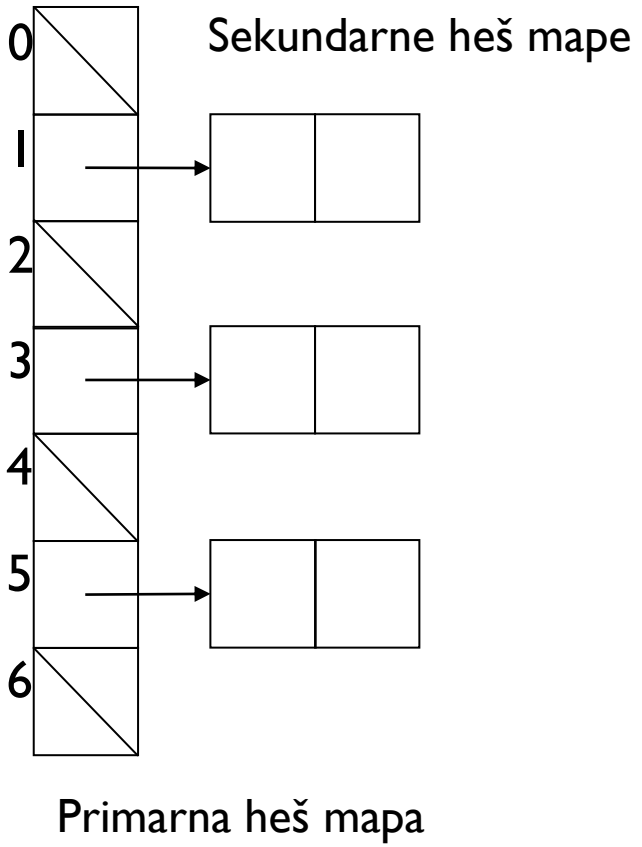
Veličina primarne heš mape = n

Veličina j-te sekundarne heš mapa = n_j^2

(gde je n_j broj ključeva koji se heširaju u lokaciju j u primarnoj heš tabeli)

Univerzalne heš funkcije u svim heš mapama

Izvodi se nekoliko slučajnih proba, dok se ne dobiju heš funkcije bez kolizija



Rešavanje kolizija – rezime

Ulančavanje sinonima

- Proširenje mape putem sekundarnih rečnika
- Dozvoljeno je $\lambda > 1$

Otvoreno adresiranje

- Pretraga unutar heš mape
- Sekundarno traženje {linearno, kvadratno, dvostruko heširanje}
- $\lambda \leq 1$ (po definiciji)
- $\lambda \leq ½$ (poželjno)

Rešavanje kolizija – rezime

Reheširanje

- Optimizacija heš mape kada λ pređe određenu granicu

Heš funkcije

- Jednostavni celobrojni heš – veličina mape prost broj
- Metod množenja
- Univerzalno heširanje garantuje da ulaz nije uvek loš

Perfektno heširanje

- Zahteva poznat i fiksiran skup ključeva
- Garantovana složenost – $O(1)$ vreme i $O(n)$ memorija

Strukturirano programiranje

Strukturirano programiranje

Kod prvih računara, programi su razvijani **ad hoc metodama** – nešto se pokuša i vidi se efekat

Kroz analizu ranih softverskih projekata, uočeni su **obrasci u programiranju** koji su **obično davali dobre rezultate** i na taj način nastala su **pravila strukturiranog programiranja**

Strukturirani programi su **lakši** za **čitanje** i **razumevanje** od nestrukturiranih, **olakšano** je i **testiranje**, **debagiranje**, **modifikacija** i **održavanje koda**, kao i **rad većih timova** programera na razvoju kompleksnijih programa

Strukturirano programiranje

E. Dijkstra, "Go To Statement Considered Harmful", *Communications of the ACM*, Vol. 11 (March1968), No. 3, pp. 147–148.

- **broj grešaka u programskom kodu je proporcionalan broju upotreba go to naredbe**

Niklaus Wirth razvio je jezik **Pascal** 1970. koji je podržao dobre prakse strukturiranog programiranja

Strukturirano programiranje je programska **paradigma** koja ima za cilj **poboljšanje jasnoće i kvaliteta,** kao i **skraćenje vremena razvoja računarskih programa**, kroz upotrebu **potprograma, blokovskih struktura** i **iteracija**

Predstavlja značajno unapređenje u odnosu na primenu jednostavnih testova i skokova (npr. **go to** naredbe) koji dovode do "špageti koda" (engl. *spaghetti code*), teškog za razumevanje i održavanje

Strukturirano programiranje

Strukturirano programiranje je skup procesa u projektovanju i implementaciji čija primena dovodi do dobro strukturiranih programa

- **disciplinovan pristup** programiranju
- **projektovanje odozgo-nadole** (engl. *top-down design*)
- **usavršavanje (preciziranje) programa u koracima** uz korišćenje **ograničenog skupa kontrolnih struktura**

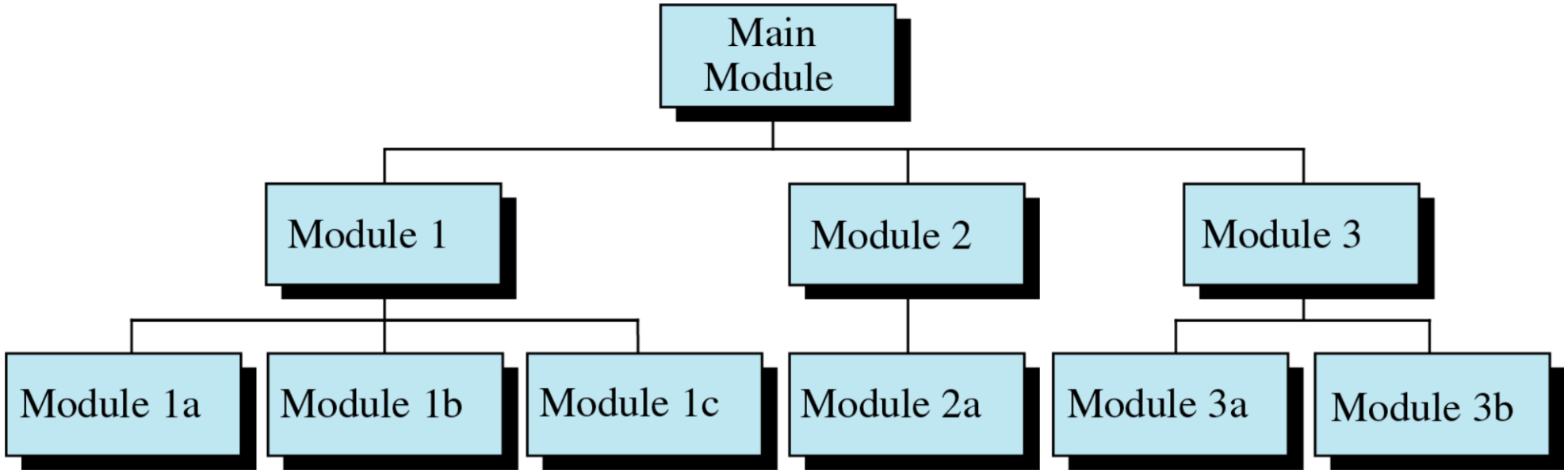
Skup kontrolnih struktura koje se koriste u strukturiranim programima:

- **sekvenca**
- **selekcija**
- **iteracija**

Projektovanje odozgo-nadole

Program je podeljen na **glavni modul** i njemu hijerarhijski **podređene module** (engl. *modules*) – **potprograme**

Svaki modul je dalje podeljen na **podmodule** (engl. *submodules*) dok se ne dobiju rezultujući moduli koji se mogu razumeti bez dalje podele



Kontrolne strukture i strukture podataka

Kontrolne strukture i **strukture podataka** su **glavne komponente programa**

Strukture podataka omogućavaju čuvanje i obradu podataka

- najprostije su konstante (7, 3.14, 'A', …)
- promenljive su jednostavne strukture podataka – imaju ime, vrednost, tip i lokaciju (adresu) u memoriji
- složenije strukture podataka su polja, spregnute liste, stabla, datoteke, itd.

Kontrolne strukture određuju redosled izvršavanja naredbi (i da li će uopšte biti izvršene) – **selekcija** i **iteracija**

Pravilni programi

Svaki **dijagram toka** (engl. *control flow* diagram) predstavlja **usmereni graf** – digraf, čiji čvorovi mogu biti:

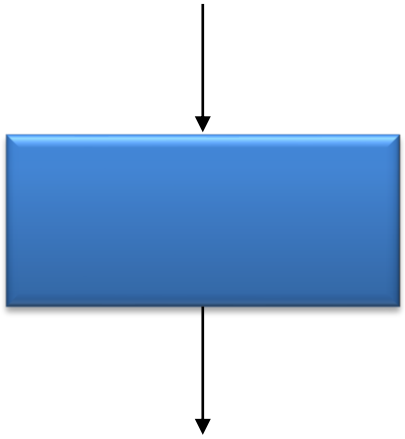
- **proces**
- **predikat**
- **kolektor**

Proces

Proces (**funkcijski čvor** ili **obrada**) je element dijagrama toka sa jednom ulaznom i jednom izlaznom linijom u kome se obavlja obrada ili prenos podataka

Najčešća realizacija **procesa** je u vidu naredbe **dodele vrednosti**

proces
ulazni stepen 1
izlazni stepen 1

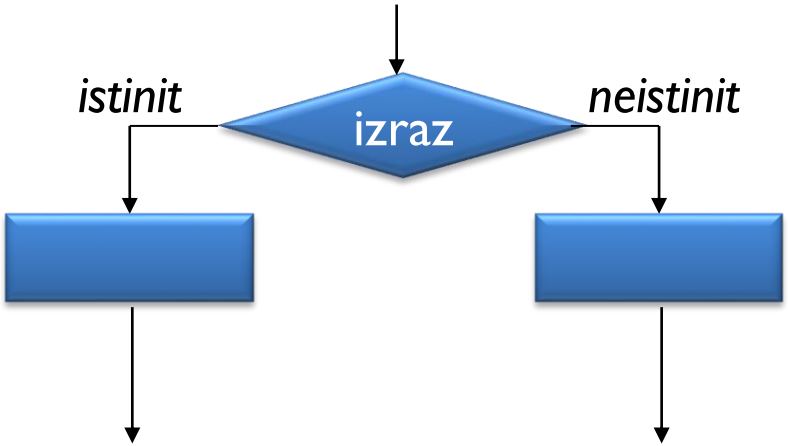


Predikat

Predikat je element dijagrama toka sa jednim ulazom i dva izlaza koji obavlja isključivo kontrolnu funkciju:

- izračunava vrednost pridruženog logičkog izraza
- potom usmerava dalji tok obrade u pravcu jedne od izlaznih linija, ako je izračunata vrednost *istina (true)*, a u pravcu druge, ako je izračunata vrednost *neistina* (*false*)

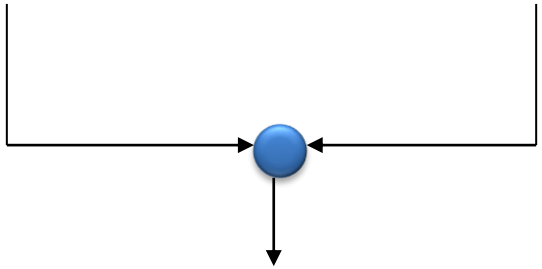
predikat
ulazni stepen 1
izlazni stepen 2



Kolektor

Kolektor je element dijagrama toka sa dve ulazne linije i jednom izlaznom linijom koji ne obavlja obradu i koji se primenjuje kada dva nezavisna toka obrade treba usmeriti u nekom zajedničkom pravcu

kolektor
ulazni stepen 2
izlazni stepen 1



Pravilan program

Pravilni program (engl. *proper program)* je program koji zadovoljava sledeća tri uslova:

- ima **tačno jednu ulaznu granu**
- ima **jednu izlaznu granu**
- **kroz svaki čvor prolazi se bar jednom od ulaza do izlaza**

Funkcija prva dva uslova jeste da se **ceo pravilni program** može **zameniti tačno jednim čvorom**, a funkcija trećeg jeste da **eliminiše mogućnost beskonačnih iteracija** i **izolovanih delova programa** (programskih segmenata)

Primeri programa sa beskonačnim iteracijama i izolovanim segmentima

```
int i = 0;
while (i <=20){
        printf("%d ", i);
        i--;
}
```

Program se izvršava dok se ne pritisne Ctrl+C ili dok ne dođe do greške potkoračenja (engl. *underflow error*)

```
int f (int x, int y) {
        return x + y;
        int z = x * y;
}
```

Naredba koja sledi iza **return** nikada neće biti izvršena, pa se može i odbaciti!

Pravilan potprogram

Svaki **sastavni deo pravilnog programa** (tj. neki podgraf u dijagramu toka) koji je **takođe pravilan program** naziva se **pravilan potprogram**

- Svi **procesi** u sastavu pravilnog programa **jesu** pravilni potprogrami
- **Predikati nisu** pravilni potprogrami jer imaju po dva izlaza
- **Kolektori nisu** pravilni potprogrami jer imaju po dva ulaza

Prost program

Pravilan program čiji svaki sastavni deo, koji je pravilan potprogram, sadrži samo jedan čvor, naziva se **prost program**

- Prosti programi se **ne mogu dalje dekomponovati** u pravilne potprograme
- **Primere prostih programa** nalazimo među **osnovnim kontrolnim strukturama**

Strukturna teorema

Pravilni programi se **uvek mogu realizovati superpozicijom tri osnovne kontrolne strukture** (**sekvence, selekcije, iteracije**)

Teorema Bohm-Jacoppini (1966):

Klasa **grafova dijagrama toka** (engl. *control flow graphs*) može **realizovati bilo koju izračunljivu funkciju kombinacijom potprograma** na **tri načina** (putem tri kontrolne strukture):

- izvršavanjem jednog potprograma, praćenog drugim potprogramom (**sekvenca**)
- izvršavanjem jednog od dva potprograma u zavisnosti od vrednosti logičkog izraza (**selekcija**)
- izvršavanjem potprograma dok god je logički izraz tačan (**iteracija**)

Može se koristiti i dodatna promenljiva u vidu bitova, kako bi se pamtile informacije o lokaciji u programu

Baza strukturiranih programa

Skup prostih programa čijom **superpozicijom** se može **realizovati proizvoljan pravilan program** nazivamo **baza strukturiranih programa**

Baza strukturiranih programa je **dovoljna**, ako se **izostavljanjem neke od kontrolnih struktura iz baze dobija neka prostija baza**, u suprotnom baza je **potrebna**

Formalno strukturiran program je **svaki pravilan program** sačinjen korišćenjem **jedino kontrolnih struktura** iz određene **baze**

Dva programa su **ekvivalentna** ako **realizuju istu obradu** podataka, tj. ako za iste ulazne podatke generišu iste izlaze

Osnovna strukturna teorema

Svaki **pravilan program** može se transformisati u **ekvivalentan formalno strukturiran program** uz korišćenje baze od **tri osnovne kontrolne strukture (sekvenca, selekcija i iteracija)**, **primenjene na procese i predikate polaznog programa,** uz **moguću primenu jedne dopunske selektorske promenljive**

pravilan program = sekvenca + selekcija + iteracija

Strukturna teorema sa dve kontrolne strukture

Svaki **pravilan program** može se transformisati u **ekvivalentan formalno strukturiran program** uz korišćenje baze od samo **dve osnovne kontrolne strukture (sekvenca i iteracija)**, **primenjene na procese i predikate polaznog programa**, uz **moguću primenu dopunskih selektorskih i logičkih promenljivih**

pravilan program = sekvenca + iteracija

Strukturna teorema sa dve kontrolne strukture

IF p THEN X ELSE Y END_IF

je ekvivalentno sa:

```
a = p ;
b = not(p) ;
WHILE a DO
        X;
        a = false;
END_WHILE;
WHILE b DO
        Y;
        b = false;
END WHILE;
```

Strukturirano programiranje – rezime

Strukturirani programi su:

- **lakši** za **čitanje** i **razumevanje** od nestrukturiranih,
- **lakši** za **testiranje**, **debagiranje**, **modifikaciju** i **održavanje koda**,
- **olakšali rad većih timova** programera na razvoju korektnih kompleksnih programa,
- **smanjili** su **troškove razvoja softvera** i
- postavili **osnove za razvoj objektno-orijentisane paradigme** koja je danas dominantna u softverskoj industriji

Testiranje programa

Razvoj programa jednostavne funkcionalnosti

1. Precizan opis problema

- Šta program treba da radi? Odgovor pruža **specifikacija**
- **IDENTIFIKOVATI PROBLEM** – koncizno opisati problem, preduslove za rad programa i očekivanja od rada programa
- **ODREDITI ULAZE I IZLAZE** – koje podatke program treba da dobije da bi obavio željenu radnju i koje podatke će program vratiti korisniku

2. Definisanje modela i izbor metode

- opis problema formalnim matematičkim aparatom
- tačne i približne metode, ocena složenosti izabranih modela i metoda
- **DEKOMPOZICIJA** - razlaganje problema na potprobleme sve do elementarnih problema koje se znaju rešiti i implementirati
- **APSTRAKCIJA** - izbor informacija o problemu koje su relevantne i ignorisanje informacija koje su irelevatne za dati kontekst

Razvoj programa jednostavne funkcionalnosti

3. Projektovanje algoritma

- postupak (način razmišljanja) kojim se iz datih polaznih podataka, određenim konačnim nizom računskih i/ili logičkih postupaka dolazi do traženih rezultata
- osobine: diskretnost, determinisanost, efektivnost, rezultativnost, generalnost
- primeri: recept, uputstvo za sklapanje, određivanje NZD...
- **nezavisan od programskog jezika**
- piše se u skladu sa modelom definisanim u ranijim fazama
- izbor odgovarajućih **apstraktnih tipova podataka** (ATP) (matematički (logički) model – skup podataka i operacija) i **struktura podataka** (fizička realizacija – memorijska i algoritamska reprezentacija)

Razvoj programa jednostavne funkcionalnosti

4. **Pisanje programa (programiranje)**
   - pisanje programa prema algoritmu (preslikavanje 1:N)
   - **program je implementacija algoritma na računaru instrukcijama konkretnog programskog jezika**
   - kompajlerski i interpreterski jezici

5. **Testiranje programa i pisanje dokumentacije**
   - izvršavanje programa nad test podacima sa ciljem da se pronađu greške, pisanje testova radi se pre, tokom i nakon razvoja programa
   - sintaksne i semantičke greške
   - pisanje dokumentacije je neophodan korak za proizvod

Testiranje programa

Testiranje programa je **proces** koji se koristi radi utvrđivanja **tačnosti**, **kompletnosti** i **kvaliteta** razvijenih računarskih programa

Na početku razvoja, **testiranje** je proces izvršavanja programa **u cilju pronalaženja grešaka**

Na kraju razvoja, **testiranje** je proces izvršavanja programa radi **demonstracije da nema grešaka** (nakon što su otklonjeni bagovi)

Nikada ne treba pristupiti testiranju sa pretpostavkom da program radi ispravno – uvek treba krenuti sa pretpostavkom da program sadrži greške i potom pokušati pronaći što je moguće više grešaka

Testiranje programa

Ne može se dokazati da program nema grešaka – **problem prekidanja** (engl. *halting problem*) – na osnovu opisa proizvoljnog računarskog programa i ulaza, odrediti da li će se program završiti u konačnom vremenu ili nastaviti da se izvršava u beskonačnost

Alen Tjuring je pokazao da se **algoritam** za **sve moguće parove ne može konstruisati**, tj. da problem nije rešiv na Tjuringovoj mašini

Testiranje je kao aktivnost **kompleksnija** od **kodiranja**, nema univerzalnog pristupa

Nakon svake **ispravke**, sledi **novo testiranje** programa

Testiranje programa

Testiranje programa je jedna od **najskupljih aktivnosti** u toku njegovog životnog ciklusa

Sprovodi se u toku prvobitne realizacije gde može da učestvuje čak sa 50% vremena, kao i u fazi korišćenja i održavanja i to prilikom svake modifikacije programa

Testiranjem se utvrđuje prvo kako program obavlja zadatak za koji je namenjen, a zatim i kako se ponaša u realnom okruženju

Testiranje se može definisati kao **provera**:

- **usklađenosti implementacije** (realizacije) programa **sa** njegovom **specifikacijom** i
- **ponašanja programa** u eksploatacionim uslovima

Testiranje programa

Provera usklađenosti odgovara osobini **korektnosti (pouzdanosti) programa** koja se definiše kao **mera u kojoj program zadovoljava specifikaciju**

Provera ponašanja u eksploatacionim uslovima odgovara **robustnosti programa** pod kojom se podrazumeva **imunost na razne poremećaje**, prvenstveno na **neregulame** ili **neočekivane ulazne podatke**

Pod **ulaznim podacima** smatraju se ne samo ulazne vrednosti već i **akcije** nad programom u najširoj interpretaciji (pritisak tastera, dejstvo mišem, itd.), pod **izlaznim podacima** smatraju se ne samo izlazne vrednosti, nego i bilo koja druga **reakcija** programa na ulaz

Testiranje programa – terminologija

U toku realizacije programa prave se **greške** (engl. *error*)

Rezultat greške pojavljuje se u programskom kodu kao **defekt** (engl. *fault*, *bug*)

U praksi se termini **greška** i **defekt** najčešće smatraju sinonimima, iako strogo gledano oni to nisu

Kada se u toku izvršenja programa naiđe na defekt dolazi do **otkaza** (engl. *failure*) koji može, ali ne mora biti momentalno primećen

Eksplicitno manifestovanje otkaza zove se **incident**

Testiranje programa – terminologija

Otkrivanje defekata u programu najčešće se obavlja na bazi nekih ulaznih vrednosti, konkretnih ili uopštenih

Element skupa ulaznih vrednosti kojim se program može pokrenuti i koji je uvršten u test naziva se **test slučaj** (engl. *test case*)

Skup svih test slučajeva naziva se **test skup** (engl. *test set*)

Pristupi testiranju:

- **konstruktivni** (dokazati da program nema grešaka)
- **destruktivni** (dokazati da program ima grešaka)

Prethodni pristupi su međusobno suprotstavljeni

Testiranje programa – preporuke

Test mora da **obuhvati kako validne i očekivane**, tako **i nevalidne i neočekivane uslove**

Test treba da obezbedi odgovor na sledeća pitanja:

- da li program radi ono za šta je namenjen?
- da li program možda radi i nešto što je nepredviđeno (i nepoželjno)?

Izbegavati ad hoc testove ("ubaci par ulaznih podataka, pa da vidimo šta će se desiti"), kao i **nedokumentovano testiranje** – takvi testovi ne mogu se ponoviti, niti daju bilo kakvu garanciju da će program u pružati korektan izlaz

Nikako **ne vršiti testiranje pod pretpostavkom** da **nema defekata**

Praksa pokazuje da je potencijalna **gustina grešaka najveća** u **segmentima koda** u kojima su **već ranije locirane greške**

Metode testiranja programa

Statičko (analiza koda i strukture programa, uključuje verifikaciju) i **dinamičko testiranje** (program se pokreće sa test slučajevima, uključuje validaciju)

Pristup kutija (engl. *box approach*) – opisuju tačku gledišta test inženjera kada pravi test slučajeve:

- **metod bele kutije** (engl. *white box approach*) – testira se interna struktura programa, a ne funkcionalnosti dostupne krajnjem korisniku
- **metod crne kutije** (engl. *black box approach*) – testiraju se funkcionalanosti dostupne krajnjem korisniku, bez ulaženja u internu strukturu programa

Metod bele kutije

Metod bele kutije (poznat i kao **testiranje u čistoj/staklenoj/transparentnoj kutiji** ili **strukturalno testiranje** – uvidom u kod) **ispituje internu strukturu** ili **rad programa, a ne funkcionalnosti** programa koje vidi krajnji korisnik

Metodom bele kutije **test slučajevi** se kreiraju **na osnovu internog pogleda na sistem** i primenom programerskih veština

Test inženjer bira ulazne vrednosti tako da **ispita različite putanje kroz programski kod** i odredi odgovarajuće izlaze

Metod bele kutije – tehnike

Testiranje API-ja – testiranje aplikacija korišćenjem javnih i privatnih programskih interfejsa aplikacija (engl. *Application Programming Interfaces - API*)

Metode ubacivanja grešaka (engl. *fault injection*) – namerno uvođenje grešaka u kod kako bi se ispitala efikasnost strategije testiranja

Pokrivanje koda (engl. *code coverage*) – kreiranje testova koji zadovoljavaju određene kriterijume pokrivenosti koda (npr. test inženjer može napraviti testove koji izazivaju da se sve naredbe u programu izvrše makar jedanput)

Metod bele kutije – pokrivanje koda

Alati za proveru pokrivenosti koda (engl. *code coverage tools*) služe za procenu kompletnosti test skupa kreiranog bilo kojom metodom. **Pokrivenost koda** kao softverska metrika može se predstaviti kao:

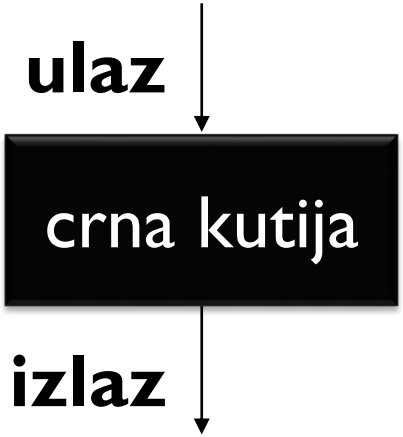
- **pokrivenost funkcija** (engl. *function coverage*) – pokazuje koje su funkcije izvršene
- **pokrivenost naredbi** (engl. *statement coverage*) – pokazuje broj linija koda izvršenih kako bi se kompletirao test
- **pokrivenost odluka** (engl. *decision coverage*) – pokazuje da li su izvršene i grana koja odgovara istini i grana koja odgovara neistini u datom testu

100% pokrivenost naredbi garantuje da su **sve putanje** ili grane (u smisli kontrole toka) u programskom kodu **izvršene najmanje jedanput.** Ovo pomaže u osiguravanju ispravnog funkcionisanja, ali nije dovoljno pošto isti kod može da obrađuje druge ulazne vrednosti tačno ili netačno

Metod crne kutije

Metod crne kutije posmatra program kao "crnu kutiju", **ispitujući njegovu funkcionalnost**, **bez** ikakvog **znanja o** njegovoj **internoj implementaciji** (bez uvida u izvorni kod programa)

Test inženjeri vide samo **šta bi program trebalo da radi**, ali **ne i kako to program radi**



Metod crne kutije

Prednost – ne zahteva poznavanje programiranja

Nedostatak – tester može da napiše puno test slučajeva kojima proverava nešto što se moglo proveriti jednim test slučajem ili napraviti test slučajeve koji ostavljaju delove programa neproverenim

Metod crne kutije može se koristiti na svim nivoima testiranja

Tehnike testiranja metodom crne kutije:

Podela na ekvivalentne klase (engl. *equivalence partitioning*)

Analiza graničnih vrednosti (engl. *boundary value analysis*)

Testiranje slučajeva korišćenja (engl. *use case testing*)

Testiranje zasnovano na specifikaciji (engl. *specification-based testing*) – provera se funkcionalnost softvera u skladu sa zahtevima, tester dobija test slučajeve i samo treba da verifikuje da li za dati ulaz program daje isti izlaz (ili ponašanje) kao što je specificirano u test slučaju

Testiranja programa – nivoi

Jedinično testiranje (engl. *unit testing*) – **provera funkcionalnosti** određenog **dela koda**, obično **na nivou funkcija**. Za jednu funkciju može postojati više test slučajeva kako bi se pokrili granični slučajevi i različita putanje kroz kod

Integraciono testiranje (engl. *integration testing*) – **provera da li su interfejsi** između delova softvera **u skladu za dizajnom softvera**. Cilj je otkriti defekte u interfejsima i interakciji između integrisanih komponenti programa (modula). Integrišu se i **testiraju progresivno veće grupe modula**, koji odgovaraju elementima arhitekture softvera, sve dok se ne obuhvati ceo sistem

Sistemsko testiranje (engl. *system testing*) – **provera kompletno rad integrisanog sistema** kako bi se verifikovalo da odgovara postavljenim zahtevima. Npr. sistemsko testiranje bi moglo uključivati proveru interfejsa za prijavljivanje (logovanje), kreiranje i izmenu stavke u programu, slanje ili prikaz rezultata, praćeno brisanjem ili arhiviranjem stavki i, na kraju, odjavljivanje iz programa

Proces testiranja programa

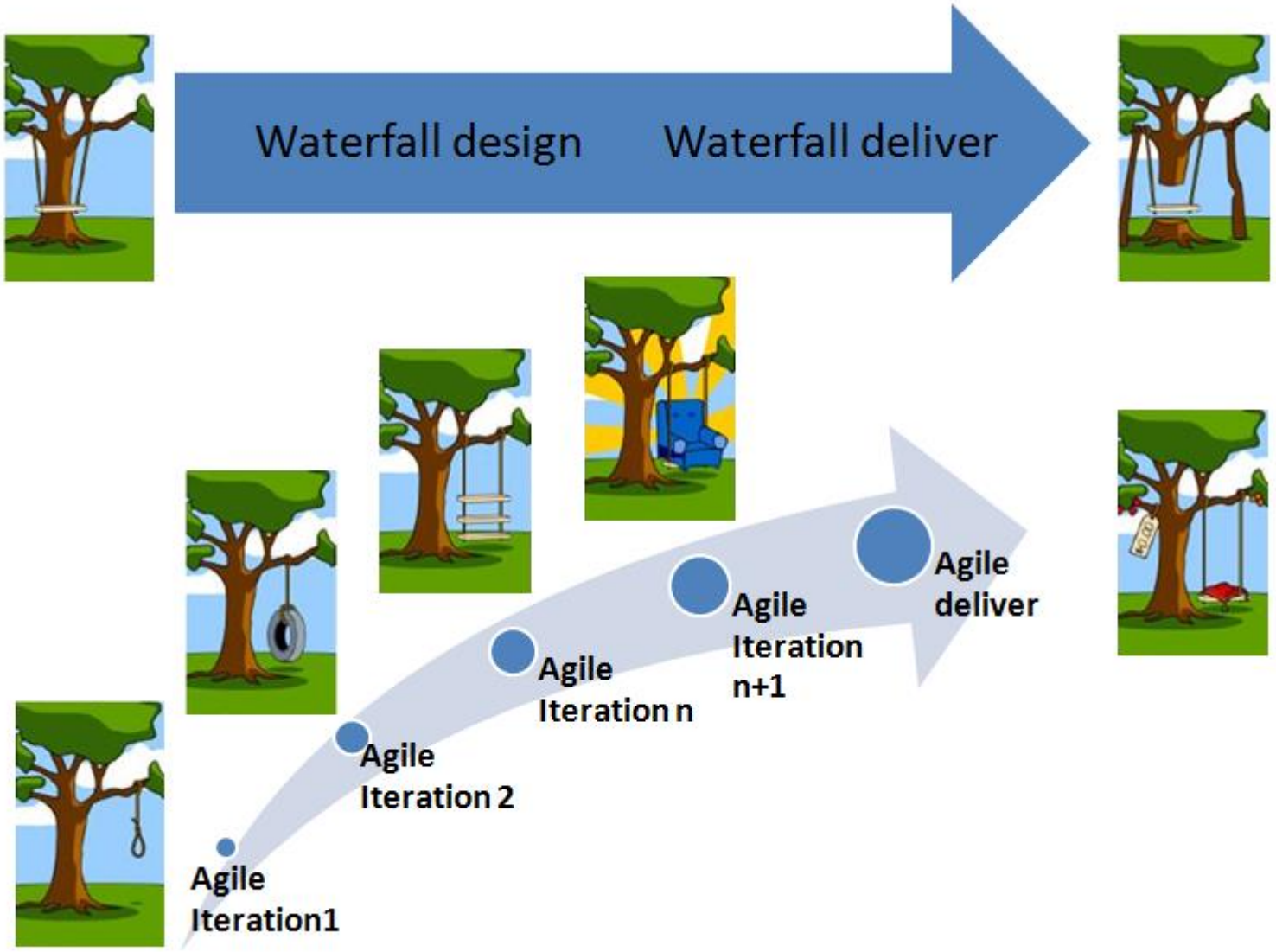
Tradicionalne metode razvoja programa – model vodopada (engl. *waterfall*)

Testiranja programa na tradicionalni način se sprovodi od strane **nezavisne grupe testera** nakon što se neka funkcionalnost implementira, a pre nego što se program isporuči klijentu. Drugi pristup je da se sa testiranjem krene čim se započne projekat i da ono bude kontinualni proces dok se projekat ne završi

Agilne metode razvoja programa (engl. *agile*)

Model razvoja softvera upravljan testovima (engl. *test-driven software development – TDD*) – prvo se pišu jedinični testovi i očekuje se da testovi inicijalno budu neuspešni. Kako se program razvija, tako prolazi sve veći deo test skupa. Test skup se kontinualno ažurira kako se otkrivaju novi granični slučajevi. Krajnji cilj je postići **kontinualnu integraciju** i **kontinualno isporučivanje** (engl. *continuous integration and continuous delivery* – **CI/CD**), tj. često objavljivanje novih verzija programa

Proces testiranja programa



Izvor: http://www.giks.org/wp-content/uploads/2015/05/Waterfall-VS-agile.png