Prevodjenje programskih jezika

$$T(n) = aT(\frac{n}{b}) + cn^k$$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{ako je } a > b^k \\ \Theta(n^k \log n) & \text{ako je } a = b^k \\ \Theta(n^k) & \text{ako je } a < b^k \end{cases}$$

Nemanja Mićović

Matematički fakultet
Oktobar 2015

Sadržaj

Zdravo svete!

Tekst pred vama je delo studenata smera Informatika sa Matematičkog fakulteta. Zahvaljujemo se puno profesoru Predragu Janičiću i profesoru Miodragu Živkoviću koju su nam svojim predavanjima i savetima umnogome pomogli da sastavimo dati tekst.

Tekst je izrađen u LaTeX- u, a ukoliko vas interesuje sam LaTeX, savetujemo da pogledate [1]. Implementacije algoritama su date u programskom jeziku C, a C je izabran iz činjenice da je to programski jezik koji se najčešće radi kada se uče osnove programiranja našim školskim ustanovama.

Smatramo da kod, bio pseudo ili pravi, treba da bude intuitivno jasan već na prvi pogled, a ne da sadrži veliku količinu slova koja označavaju razne pojmove i da korisnik provede trećinu svoga vremena samo u pokušaju da razume šta označava šta. Samim tim, trudili smo se da naše implementacije, kodovi i zapisi rešenja budu izuzetno jednostavni i da korisniku što pre predstave rešenja. Rešenja mnogih algoritama koje budemo dali mogu se dodatno optimizovati, ali proces optimizacije algoritama nije tema kojom ćemo se baviti u ovom tekstu. Naš glavni cilj jeste da da rešenja budu tačna, jasna i asimptotski u traženim okvirima.

Ukoliko imate bilo kakve sugestije ili pitanja, možete nas kontaktirati na algoritmiNLN@gmail.com.

Autori

POGLAVLJE 1

Elementi teorije formalnih jezika

U narednom poglavlju, dajemo progled osnovnih pojmova u toeriji formalnih jezika. Razjasnicemo pojmove kao sto su jezik, azbuka, regularni jezik i slicno.

1.1 Azbuka i jezik

Definicija 1.1.1 *Azbuka (alfabet), u oznaci $\sum$, je konacan skup simbola koje nazivamo karakteri.*

Primer 1.1.1 $\sum = \{a, b, c\}$ *je jedna azbuka.*

Definicija 1.1.2 *Rec (niska) je bilo kakav konacan niz karaktera azbuke.*

Primer 1.1.2 *Neka je data azbuka:* $\sum = \{a, b, c\}$.
$x = ab \in \sum$
$y = abcaabc \in \sum$

Definicija 1.1.3 *Skup svih reci nad azbukom $\sum$, u oznaci $\sum^*$, definisemo kao skup koji sadrzi sve moguce reci koje se mogu konstruisati nad azbukom $\sum$.*

Sa ϵ cemo oznacavati praznu rec.

Primer 1.1.3 *Neka je data azbuka* $\sum = \{a\}$.
Tada je skup $\sum^* = \{\epsilon, a, aa, aaa, aaaa, ...\}$.
Primetimo da ce skup $\sum^*$ biti beskonacan u opstem slucaju.

Definicija 1.1.4 *Skup $L \subseteq \sum^*$ nazivamo jezik.*

Primer 1.1.4 $L_1 = \{a, ba, bba, bbba, bbbba, ...\}$
Primetimo da L_1 mozemo i zapisati kao $L_1 = \{b^k a \mid k \geq 0\}$.
Obrnuto, neka je dat jezik $L_2 = \{a^n b^n \mid n > 0\}$
Tada vazi $L_2 = \{ab, aabb, aaabbb, ...\}$.

Ono sto je kljucni problem kojim cemo se baviti na dalje, jeste problem ispitivanja pripadnosti reci jeziku.

1.2 Operacije nad jezicima i recima

Operacije nad recima

Uvedimo prvu operaciju - konkatenacija, odnosno spajanje reci. Cesto se srece operator . koji vrsi konkatenaciju reci[1]. Takodje, koristicemo i operator $\cdot$ za oznacavanje uvedenog operatora.

Teorema 1.2.1 *Algebarska struktura* $(\sum^*, \cdot)$ *je monoid.*

Dokaz 1.2.1 *Neka su* $x,\ y,\ z \in \sum^*$ *neke reci azbuke* $\sum$.
Znamo da $\exists \epsilon \in \sum^*$ *iz cega sledi da* $x \cdot \epsilon = \epsilon \cdot x = x$ *cime je pokazano svojstvo neutrala u odnosu na* $\cdot$.
Neka je $x \cdot y = w \to w \in \sum^*$ *jer je* $\sum^*$ *skup svih reci nad azbukom* $\sum$.
I konacno, pokazimo svojstvo asocijativnosti nad recima x, y, z. No to je ocigledno jer:
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$ *te je zaista* $(\sum^*, \cdot)$ *monoid.*

Primetimo da $(\sum^*, \cdot)$ nije komutativan jer u opstem slucaju $ab \neq ba$, ako su ab i ba reci iz $\sum^*$.

Operacije nad jezicima

Primetimo da smo jezik definisali kao skup odakle prirodno slede skupovne operacije. Time dajemo sledecu definiciju.

Definicija 1.2.1 *Neka su dati jezici* $L, L_1, L_2 \subseteq \sum^*$. *Skupovne operacije* $\cup, \cap, \setminus$ *i* ^{c} *definisemo kao:*

- $L_1 \cup L_2 := \{x \mid x \in L_1 \vee x \in L_2\}$
- $L_1 \cap L_2 := \{x \mid x \in L_1 \wedge x \in L_2\}$
- $L_1 \setminus L_2 := \{x \mid x \in L_1 \wedge \notin L_2\}$
- $L^c := \sum^* \setminus L$
- $L_1 \times L_2 := \{(x, y) \mid x \in L_1, y \in L_2\}$
- $L_1 \cdot L_2 := \{x \cdot y \mid x \in L_1 \wedge y \in L_2\}$

[1] Programski jezik PHP koristi operator . kao operator konkatenacije stringova.

$L_1 \cdot L_2$ cemo nazivati *proizvod jezika.* Primetimo i sledece:

$$L^0 = \{\epsilon\}$$

$$L^n = L \cdot L^{n-1} = \prod_{i=1}^{n} L_i$$

Da li vazi asocijativnost?

$$L^3 = L \cdot \Big(L \cdot \big(L \cdot \{\epsilon\}\big)\Big)$$

$$L^3 = \Big(\big(\{\epsilon\} \cdot L\big) \cdot L\Big) \cdot L$$

Cime smo problem sveli na problem rasporedjivanja zagrada, a za to naravno vazi asocijativnost.

$$\Longrightarrow L \cdot L^{n-1} = L^{n-1} \cdot L$$

Primer 1.2.1 $\{ab, b\}^3 = \{ab, b\}\{ab, b\}^2$
$\{ab, b\}^3 = \{ab, b\}\{abab, abb, bab, bb\}$
$\{ab, b\}^3 = \{ababab, ababb, abbab, abbb, babab, babb, bbab, bbb\}$

Primetimo da smo od 2 reci na 3. stepen dosli do skupa kardinalnosti 8. Postavlja se pitanje da li u opstem slucaju je $|L_1 \cdot L_2| = |L_1| \cdot |L_2|$?

Primer 1.2.2 $\{ab, a\} \cdot \{b, \epsilon\} = \{abb, ab, a\}$
$|\{abb, ab, a\}| = 3 \neq 4 \Longrightarrow |L_1 \cdot L_2| \neq |L_1| \cdot |L_2|$

POGLAVLJE 2

Matematička indukcija

U ovom poglavlju upoznaćemo se sa jednim od moćnih matematičkih alata za dokazivanje tvrđenja. Indukcija ima ogromnu primenu u svetu matematike i može se iskoristi za veliki broj dokaza, pogotovo rekurzivnih funkcija u računarstvu.

Spojiti sa poglavljem SLOZENOST ALGORITAMA

POGLAVLJE 3

Složenost algoritama

Upoznaćemo se sa notacijama O, Ω i Θ kojima ćemo izražavati složenost algoritama. Pokazaćemo tehnike za određivanje složenosti raznih algoritama i daćemo prikaz i osnovne informacije i klasa složenosti.

Spojiti sa poglavljem MATEMATICKA INDUKCIJA

Kao što često u životu postoji više tačnih odgovora na dato pitanje, tako u svetu programiranja postoji više mogućih rešenja za predstavljeni problem. Upravo u tome i jeste lepota programiranja i glavni argument ljudi koji veruju da je programiranje umetnost. Činjenica da programer može da izrazi svoju kreativnost kroz tekst i da ukoliko se potrudi uvek taj tekst napiše drugačije, samom programiranju daje na jedinstvenosti i lepoti. I kada razmislimo malo bolje o tome, činjenica da ljudi danas masovno koriste softver, svakako češće nego što čitaju knjige, gledaju filmove i slike, koji su opšte prihvaćeni vidovi umetnosti, govori o tome koliko je danas programiranje bitno i svuda oko nas. Ono što možda ostaje nedorečeno, jeste na primer, da prosečan korisnik pametnog telefona zapravo i ne zna kako uopšte njegov telefon funkcioniše i ko to uopšte pravi, što svakako daje umnogome različit odgovor od toga ko kreira filmove, slike i knjige.

Algoritmi

POGLAVLJE 4

Elementarne strukture podataka

U ovom poglavlju razjasnićemo pojmove poput: lista, heš tablela, stablo, stek, red i slično, i dati uvod u osnovne i najčešće fundamentalne strukture podataka. Takođe, daćemo i algoritme koji implementiraju osnovne naredbe koje određene strukture podataka imaju.

NAPOMENA

Ovo poglavlje bi trebalo da pomene stabla, ali se stabla obrađuju u poglavlju koje se bavi grafovima.

4.1 Osnovne strukture podataka

4.2 Povezane liste

Jednostruko povezane liste

Dvostruko povezane liste

Kružne liste

4.3 Stek

Stek (eng. stack) je struktura podataka koja je zasnovana na principu LIFO - Last in first out. To jest, poslednji element koji je dodat na stek se prvi skida sa steka. Stek ima sledeće osnovne naredbe:

- Push - gurni element na vrh steka
- Pop - skini element sa vrha steka
- Top - pogledaj šta je na vrhu steka

Stek se može realizovati na više načina, a najčešće je to preko niza i povezanih lista. Objektni programski jezici često imaju sistemsku klasu za rad sa stekom.

Primer steka na računaru je sistemski stek na koji se stavljaju pozivi funkcija i tako dalje. Greška *stack overflow* je greška prekoračenja steka, to jest, greška koja se javlja kada se pokuša naredba *push* a stek je pun, ili naredba *pop*, a stek je prazan.

Za rad sa stekom potrebno je čuvati podatak o vrhu steka i paziti na ograničenje elemenata kako se ne bi došlo do pomenute greške *stack overflow*.

Složenost naredbi:

- Push - O(1)
- Pop - O(1)
- Top - O(1)

Naredba steka su konstante složenosti jer mi u svakom trenutku znamo gde je vrh steka, tako da u konstatnom vremenu možemo da izvršimo potrebnu operaciju.

4.4 Red

Red (eng. queue) je struktura podatak zasnovana na principu FIFO - First in first out. To jest, prvi element koji je dodat se i prvi skida. Najlakše je zamisliti red i realnog života, na primer red ljudi koji čeka kod doktora. Doktor prima osobu koje je prva u redu, to jest, osobu koja je prva došla. Čovek koji poslednji dođe, ide na kraj reda.

Red ima sledeće osnovne naredbe:

- Add - dodaj element na kraj reda
- Remove - skini element sa početka reda

Obe operacije su složenosti O(1) jer se u svakom trenutku čuvaju podaci o tome gde se nalazi početak reda, a gde kraj reda.

Implementacija reda pomoću dva steka

Ako su nam data dva steka S_1 i S_2 možemo od njih napraviti red Q. Ideja je....

4.5 Heš tabela

Heš tabele su struktura podataka koja se najčešće koristi za skladištenje određenih informacija kojima često pristupamo i gde nam je potrebno da im pristupimo u vrlo kratkom vremenskom intervalu. To se postiže takozvanim heširanjem (eng. hashing) koje se vrši tako što se element x koji se smešta u heš tabelu šalje kao argument funkciji koja vrši heširanje $h(x)$.

Najveći problem sa heš tabelama jeste izabrati odgovarjuću heš funkciju koja će da elemente što ravnopravnije distribuira kroz tabelu, to jest da minimizira *kolizije*.

Kolizije su slučaj kada se desi da je pozicija i u tabeli H zauzeta, te je potrebno odabrati način na koji će se pronaći pozicija j na koje će se smestiti element x.

Dobrim odabirom heš funkcije, može se dobiti složenost O(1) za operacije heš tabele.

Operacije heš tabele su:

- Add - dodaj element u heš
- Remove - ukloni element iz heša (ukoliko postoji)
- Find - pronađi element u hešu (ukoliko postoji)

U idealnom slučaju su ove operacije složenosti O(1) ukoliko heš funkcija izbegava kolizije. Praktično, to je skoro nemoguće te uvek težimo da heš funkcija prouzrokuje što manje kolizija.

Ne postoji direktan odgovor na pitanje *Kako kreirati dobru heš funkciju* jer njen odabir zavisi od razloga zbog kojega se koristi heš tabela.

Generalno je praksa da ukoliko je popunjenost heš tabele oko 70% treba proširiti heš.

Lančanje

Nizanje

Duplo heširanje

POGLAVLJE 5

Sortiranje

Sortiranje se jako često vrši u svetu programiranja, a postoji veliki broj algoritama za sortiranje. U ovom poglavlje daćemo neke najpoznatije, analiziraćemo njihovu složenost i efikasnost, dokazati korektnost, a takođe ćemo dati i njihovu implementaciju u pseudo kodu, ali i u programskom jeziku C.

5.1 Sortiranje umetanjem - Insertion sort

5.2 Sortiranje izborom - Selection Sort

5.3 Sortiranje objedinjavanjem - Merge Sort

Priču o prvom divide and conquer algoritmu počećemo na nešto drugačiji način.

Pretpostavimo da imamo n vojnika, a da neprijateljski vojskovođa ima m vojnika, pri čemu važi da je m > n, to jest, neprijateljske vojske je više, i važi da nema povlačenja iz bitke. Ukoliko bi naš vojskovođa napao odjednom m vojnika kojih je više, imao bi mnogo veće gubitke, i vrlo verovatno bi svi izginuli. Sa druge strane, ukoliko je naš vojskovođa izuzetno dobar taktičar, a njegova vojska izuzetno utrenirana i iskusna, deljenjem neprijateljskih m vojnika na podvojske, naš vojskovođa bi bio u stanju da čak i brojčano nadjača neprijatelja i porazi svih m vojnika u kranjem ishodu.

Upravo se ovde krije ideja divide and conquer algoritama kojima pripada sortiranje objedinjavanjem. Delimo dati niz na dva podniza jednake (ili za 1 različite) dužine, rekuzivno ih sortiramo, a potom ih spajamo u jedan sortirani niz.

Pseudo kod algoritma:

```
Algoritam: MergeSort(A, p, r)
Ulaz: A, p, r
Izlaz: A (sortiran)
BEGIN
  if(p < r)
    q = (p + r) / 2 // zaokruzujemo na manji
    MergeSort(a, p, q);
    MergeSort(a, q + 1, r)
    Merge(a, p, q, r);
END
```

Pri čemu algoritam koji vrši objedinjavanje ima sledeći pseudo kod:

```
Algoritam: Merge(A, p, q, r)
Ulaz: A, p, q, r
Pomocno: X, Y // pomocni nizovi
Izlaz: A(objedinjen deo od A[p]...A[q] i A[q+1]...A[r])
BEGIN
  // kopira deo A[p]..A[q] u X[0]...X[q - p + 1]
  // n je duzina niza X
  n = KreirajNiz(A, X, p, q, r);
  // kopira deo A[q+1]...A[r] u Y[0]...[r - q]
  // m je duzina niza Y
  m = KreirajNiz(A, Y, p, q, r);
  i := 0
  j := 0
```

```
  k := 0
  while k < n + m do
    if i = n
      for i = j to m
        A[k] = Y[i]
    else if j == m
      for j = i to n
        A[k] = X[j]
    else if X[i] < Y[j]
      A[k] = X[i]
      i := i + 1
    else
      A[k] := Y[j]
      j := j + 1
    k := k + 1
END
```

Implementacija u C-u:

```
void mergeSort(int *a, int p, int r)
{
  int q;
  if(p < r){
    q = (p + r) / 2; // racunamo sredinu
    mergeSort(a, p, q);
    mergeSort(a, q + 1, r);
    merge(a, p, q, r);
  }
}
```

Dokaz

Analiza složenosti

5.4 Brzo sortiranje - Quick Sort

Mnogi tvrde daje Quick sort algoritam najkorišćeniji na svetu.

ovde malo istorijat

ovde sada kako ide algoritam kroz tekst

Dalje sledi implementacija algoritma u programskom jeziku C. Savetuje se kreiranje omotača kako bi korisnik funkcije oslobodio brige o tome koje granice niza da prosledi.

```
void quickSort(int *a, int n)
{
  quickSort_(a, 0, n-1);
}
```

```c
void quickSort_(int *a, int levo, int desno)
{
  int pivot;
  if(levo < desno){
    pivot = particionisanje(a, levo, desno);
    quickSort_(a, levo, pivot - 1);
    quickSort_(a, pivot + 1, desno);
  }
}
```

Najteži deo algoritma, korak particionisanja se može uraditi na više načina. Jedna konkretna implementacija data je u delu teksta koji sledi.

```c
1  int particionisanje(int *a, int p, int q)
2  {
3    int pivot, levo, desno;
4    pivot = a[p]; // biramo prvi element za pivota
5    levo = p + 1; // levo indikator na pocetku (posle pivota)
6    desno = q; // desni indikator na kraju niza
7
8    while(levo <= desno){
9      while(a[levo] <= pivot && levo <= q)
10       levo++;
11     while(a[desno] > pivot && desno > p)
12       desno--;
13     if(levo < desno)
14       razmeni(a, levo, desno);
15   }
16 /*Pivot je na pocetku niza, a
17 ostatak je poredjan tako da je na poziciji
18 'desno' element koji je manji od pivota
19 i on je poslednji takav u nizu, te ga menjamo sa pivotom */
20   razmeni(a, p, desno);
21   return desno;
22 }
```

Naredna implementacija particionisanja radi na sledeći način:

- Za pivota biramo prvi element u nizu (linija 4)
- Indeksi i i j kreću od prvom narednog elementa (linije 5,6)
- Indeks j ide do kraja niza (linija 6)
- Indeks i će označavati najbliži veći element od pivota
- Kada se pronađe manji element od pivota, vrši se razmena i pomera i (linije 8,9,10,11)

- Nakon petlje, treba još postaviti pivota na svoju poziciju, pri čemu znamo da $a[i-1] < a[p]$ (linija 14)

```
1 int Particionisanje(int *a, int levo, int desno)
2 {
3   int i, j, p;
4   p = a[levo];
5   i = levo + 1;
6   for(j = levo + 1; j <= desno; j++)
7   {
8     if(a[j] < p)
9     {
10      swap(a, i, j);
11      i++;
12    }
13  }
14  swap(a, levo, i - 1);
15  return i - 1;
16 }
```

Dacemo i iterativnu implementaciju.

Dokaz

Tvrđenje 5.4.1 *Algoritam QuickSort uspešno sortira niz dužine n.*

Dokaz 5.4.1 *Dokaz izvodimo potpunom indukcijom po dužini niza n.*

Baza*: Niz dužine 1 je sortiran te poziv QuickSort(a, 1) uspešno sortira niz.*

Induktiva hipoteza*: Pretpostavimo da algoritam QuickSort uspešno sortira niz dužine k, gde je $k < n$.*

Podsetimo se koraka particionisanja. Particionasanje bira pivota na zadati način i nakon završetka particionisanja važi da je pivot na poziciji p, levo od njega su elementi niz manji od pivota, a desno od njega elementi niza veći od pivota. Takođe, pivot je na svojoj poziciji. Neka su dva dobijena podniza levo i desno od pivota, dužine k_1 i k_2, respektivno. Tada važi da je $k_1, k_2 < n$. Štaviše, važi i $k_1 + k_2 < n$ jer je ukupna dužine $n-1$ (jer ne računamo pivota).

Induktivni korak*: Odnosno, algoritam QuickSort se poziva dva puta za nizove dužine k_1 i k_2, a po induktivnoj hipotezi algoritam QuickSort uspešno sortira nizove dužine manje od n.*

Analiza složenosti

Prikazaćemo najbolji i najgori slučaj, a potom dati jasnu diferencnu jednačinu algoritma.

Najgori slučaj: Najgori slučaj za algoritam predstavlja situacija kada je niz sortiran rastuće, a za pivota se uzima prvi element u nizu. Tada je složenost $O(n^2)$. Nakon dobijanja pozicije pivota i rekurzivnih poziva, dimenzija problema se smanjila samo za 1, odnosno n puta rešavamo problem čija se dimenzija smanjuje samo za 1.

Složenost najbolje slučaja: Ako pretpostavimo da ćemo uvek pivot izabrati idealno, odnosno tako da on deli niz n na 2 podniza dužine $n/2$, tada diferencna jednačina odgovara algoritmu MergeSort, odnosno:

$$T(n) = 2T(n/2) + O(n)$$

Što po Master teoremi daje složenost $O(n \log n)$

Prosečna složenost Verovatnoća da se pivot u nizu dužine n nalazi na poziciji i je $1/n$. Broj poređenja je $(n-1)$.

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$
$$T(n) = (n-1) + \frac{1}{n}(T(0) + T(1) + ... + T(n-1)) + (T(n-1) + T(n-2) + ... + T(0))$$
$$T(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}T(i) \qquad /n$$
$$nT(n) = n(n-1) + 2\sum_{i=1}^{n-1}T(i)$$

Potom se doda jos jedna jednačina za $n+1$ a onda vršimo oduzimanje:

$$(n+1)T(n+1) = (n+1)n + 2\sum_{i=1}^{n}T(i)$$
$$nT(n) = n(n-1) + 2\sum_{i=1}^{n-1}T(i)$$

Potom se vrši oduzimanje:

$$(n+1)T(n+1) - nT(n) = 2T(n) + 2n$$
$$(n+1)T(n+1) - (n+2)T(n) = 2n \qquad / \frac{1}{(n+1)(n+2)}$$
$$\frac{T(n+1)}{n+2} - \frac{T(n)}{n+1} = \frac{2n}{(n+1)(n+2)}$$

Uzmimo:

$$t_{n+1} = \frac{T(n+1)}{n+2}$$
$$t_n = \frac{T(n)}{n+1}$$

Preći ćemo na indeks i radi slobode korišćenja n kao gornje granice za sumu kasnije:

$$t_{i+1} + t_i = \frac{2i}{(i+1)(i+2)} \qquad / \sum_{i=1}^{n-1}$$
$$t_2 - t_1 + t_3 - t_2 + ... + t_n - t_{n-1} = \sum_{i=1}^{n-1} \frac{2i}{(i+1)(i+2)}$$
$$t_n - t_1 = \sum_{i=1}^{n-1} \frac{2i}{(i+1)(i+2)}$$

POGLAVLJE 6

Numerički algoritmi

Prikazaćemo osnovne i najčešće korišćene numeričke algoritme. Izračunavanje vrednosti polinoma u tački, aproksimacija nula funkcije, brzo stepenovanje, Karacuba množenje i tako dalje.

POGLAVLJE 7

Grafovski algoritmi

U ovom poglavlju ćemo se upoznati sa tim šta je graf, gde, zašto i kako se koristi. Prikazaćemo najosnovnije algoritme potrebne za rad sa grafovima i prikazaćemo kako se graf može reprezentovati u računaru. Za kraj, prikazaćemo i konkretnu implementaciju u programskoj jeziku C.

7.1 Osnovno o grafu

Definicija 7.1.1 *Graf $G = (V, E)$ je uređeni par koji sadrži skup čvorova V, i skup grana E. Grana $e \in E$ je definisana sa tačno dva čvora $v_1, v_2 \in V$.*

Grafovi su posebna tema za sebe, i ukoliko vas interesuje više o njima, možete pogledati knjige [2] i [3].

7.2 Kako reprezentovati graf u računaru?

Matrica susedstva

Liste povezanosti

7.3 Obilasci grafa

POGLAVLJE 8

Geometrijski algoritmi

U ovom delu, daćemo elementarne geometrijske algoritme koji se danas često koriste. Aproksimacije Bezijerove krive, rastojanje tačke od prave i slično. Neke od njih daćemo u pseudo kodu sa obzirom na to da C nije najpogodniji programski za bilo kakvu grafiku, dok ćemo deo algoritama zasnovanih na analitici prikazai u C-u.

POGLAVLJE 9

Stabla

Stabla predstavljaju jedan od najčešće korišćenih struktura podataka i pomoću stabala se mogu opisati raznorazni problemi i pojmovi. Zapis matematičkog izraza, rekuzivno stablo, stablo pretrage, poredbeno stablo, stablo za ilustraciju backtrack-inga, i tako dalje.

9.1 Obilasci stabla

Obilaski stabla se generalno dele u 4 grupe:

- Prefiksni - KLD
- Infiksni - LKD
- Postfiksni - LDK
- Po nivoima

Prefiksni obilazak (eng. preorder, Node-Left-Right NRL) predstavlja obilazak u kome se prvo obilazi čvor stabla, potom levo podstablo, a nakon toga desno podstablo.

Infiksni obilazak (eng. inorder, Left-Node-Right LNR) predstavlja obilazak u kome se prvo posećuje levo podstablo, potom koren, a nakon toga desno podstablo.

Postfiksni obilazak (eng. postorder, Left-Right-Node LRN) je obilazak u kome se prvo posećuju levo i desno podstablo, a potom koren.

Obilazak po nivoima je obilazak stabla u kojem se obilazi prvi nivo na kome je koren, potom nivo ispod korena sa leva na desno, i tako sve do kraja stabla, to jest, poslednjeg nivoa.

Prefiksni obilazak stabla - KLD

```
Algoritam: KLD
Ulaz: cvor (koren stabla)
Izlaz: izlazna sekvenca preorder obilaska
BEGIN
  print(cvor)
  KLD(cvor.levo)
  KLD(cvor.desno)
END
```

Infiksni obilazak stabla - LKD

```
Algoritam: LKD
Ulaz: cvor (koren stabla)
Izlaz: izlazna sekvenca preorder obilaska
BEGIN
  KLD(cvor.levo)
  print(cvor)
  KLD(cvor.desno)
END
```

Postfiksni obilazak stabla - LDK

```
Algoritam: LDK
Ulaz: cvor (koren stabla)
Izlaz: izlazna sekvenca preorder obilaska
BEGIN
  LDK(cvor.levo)
  LDK(cvor.desno)
  print(cvor)
END
```

Nerekurzivni obilasci stasbla

Preorder (KLD) obilazak stabla

Kako bismo uklonili rekurziju, biće potrebno korišćenje pomoćnog steka. Ideja je da na steku držimo sinove čvora koji ćemo prethodno skinuti i ispisati. Pseudo kod je sledeći:

```
Algoritam: KLD
Ulaz: cvor, stek
Izlaz: izlazna sekvenca preorder obilaska
BEGIN
  push(cvor);
  while(!prazan(stek)){
    x = pop(stek);
    print(x);
    if(x.desno != NULL)
      push(x.desno);
    if(x.levo != NULL)
      push(x.levo);
  }
END
```

POGLAVLJE 10

Algoritamske strategije

Algoritamskih strategija je mnogo, a u ovom poglavlju ćemo pokriti one najpoznatije i ilustrovati ih kroz primere. Razjasnićemo pojmove kao što su backtrack, divide and conquer*,* brute force*, i tako dalje.*

10.1 Osnovne algoritamske strategije

10.2 Pohlepni algoritmi

10.3 Divide and Conquer algoritmi

10.4 Backtracking

10.5 Brute force

POGLAVLJE 11

Razni zadaci

U ovom poglavlju se nalaze raznorazni zadaci koji obuhvataju tekst koje se nalazi pred vama. Nema smisla baviti se algoritmima ukoliko se oni ne implementiraju kroz programski jezik jer se nekad mogu javiti raznorazni problemi kojih tek kasnije postajemo svesni.

Bibliografija

[1] Predrag Janičić, Aleksandar Samardžić i Goran Nenadić, *LaTeX 2ε za autore*. Kompjuter biblioteka, 2003.

[2] James Anderson, *Discrete Mathematics with Combinatorics* Second Edition, 2003.

[3] D. Stevanović, M. Ćirić, S. Simić, V. Baltić, *Diskretna Matematika - Osnovi kombinatorike i teorije grafova* Društvo matematičara Srbije, 2008.

[4] Predrag Janičić i Filip Marić, *Osnove programiranja kroz programski jezik C*. Matematički fakultet, 2014.

[5] Predrag Janičić i Filip Marić, *Osnove programiranja kroz programski jezik C - Deo II*. Matematički fakultet, 2014.

[6] Predrag Živković, *Algoritmi*. Matematički fakultet, 2000.

[7] Thomash H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms*. The MIT press, III edition.

[8] R. D. Tennent *Specifzing Software*. Cambridge University Press, 2002.