Strukture podataka
Letnji semestar

Niz
(String)

Prof. Dr Leonid Stoimenov
Katedra za računarstvo
Elektronski fakultet u Nišu

Niz - Pregled

- Definicija niza
- String tip podataka
- Memorijska reprezentacija
- Operacije

Termini

Definicija: **Niz** je **sekvenca** od nula ili više **znakova** (karaktera)

- Dužina niza: Broj znakova (objekata) u nizu
- Prazan (nulti niz): Niz dužine 0
- **Indeks**:
  Pozicija znaka (objekta) u nizu
  - Indeks uzima vrednosti 0,1,2,… ili 1,2,3,…

Primeri niza

- *“”* – prazan niz
- *“Studenti”* – niz dužine 8

  indeks znaka ***S*** je *0*, a znaka ***d*** je *3*, *ako indeksiranje ide od 0*
- *“Ružica je lepo ime”* – niz dužine 17

String literal

- String literal je notacija za predstavljanje vrednosti niza unutar teksta nekog računarskog programa

- Forma za takvu notaciju je različita u programskim jezicima
- Najčešća forma je "danas je četvrtak"
- Neki jezici dopuštaju 'danas je četvrtak'
- Postoje i drugi stilovi

STRING TIP PODATKA

- Gotovo svi programski jezici imaju na neki način implementiran **string** tip podatka
- Postoje dva string tipa podataka:
  - String fiksne dužine: ima fiksiranu maksimalnu dužinu
  - String promenljive dužine: dužina nije fiksirana, ograničena je veličinom raspoložive memorije
  - Stringovi u modernim prog. jezicima su promenljive dužine
- Svaki znak niza se memoriše u jednom bajtu, a kodira se ASCII ili EBCDIC (IBM mainframe sistemi), ili **Unicode** (obično UTF-8 ili UTF-16)

Memorijska reprezentacija niza

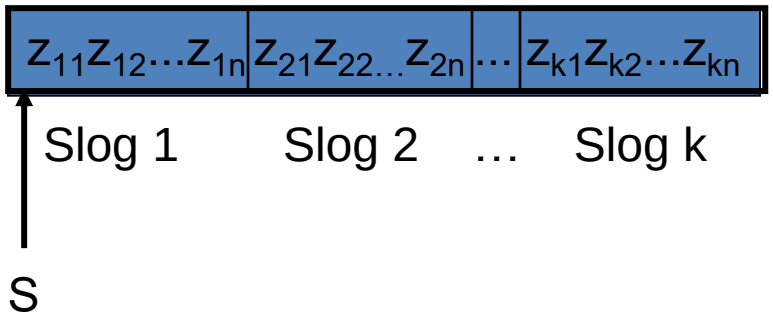
- Niz se može predstaviti
  - sekvencijalno ili
  - lančano

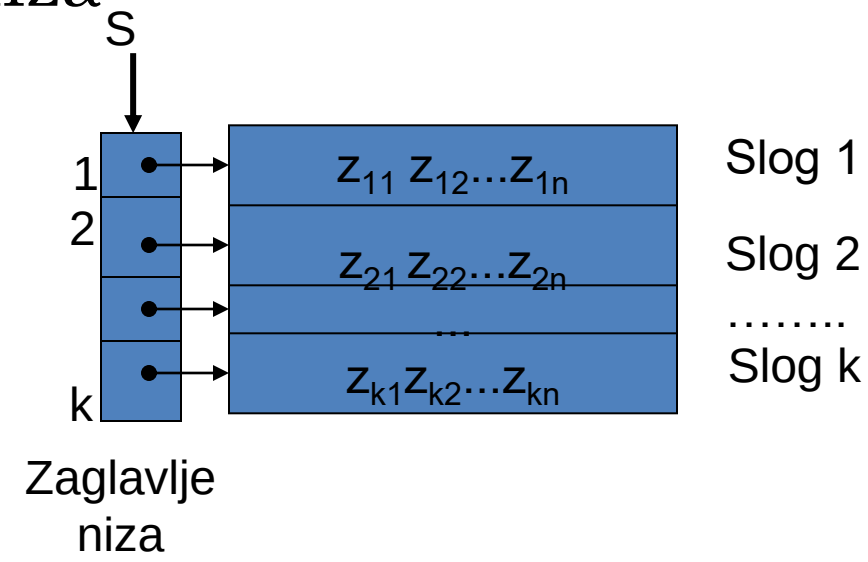
Sekvencijalna reprezentacija niza

- Niz se deli u **podnizove** i svaki podniz se posmatra kao jedan slog
- Slogovi se memorišu sekvencijalno
- Slogovi mogu biti
  - fiksne dužine ili
  - promenljive dužine

Sekvencijalna reprezentacija niza

- Slogovi fiksne dužine
  - Adresira se samo prva lokacija
  - Ako se želi referenciranje na svaki slog, uvodi se zaglavlje niza

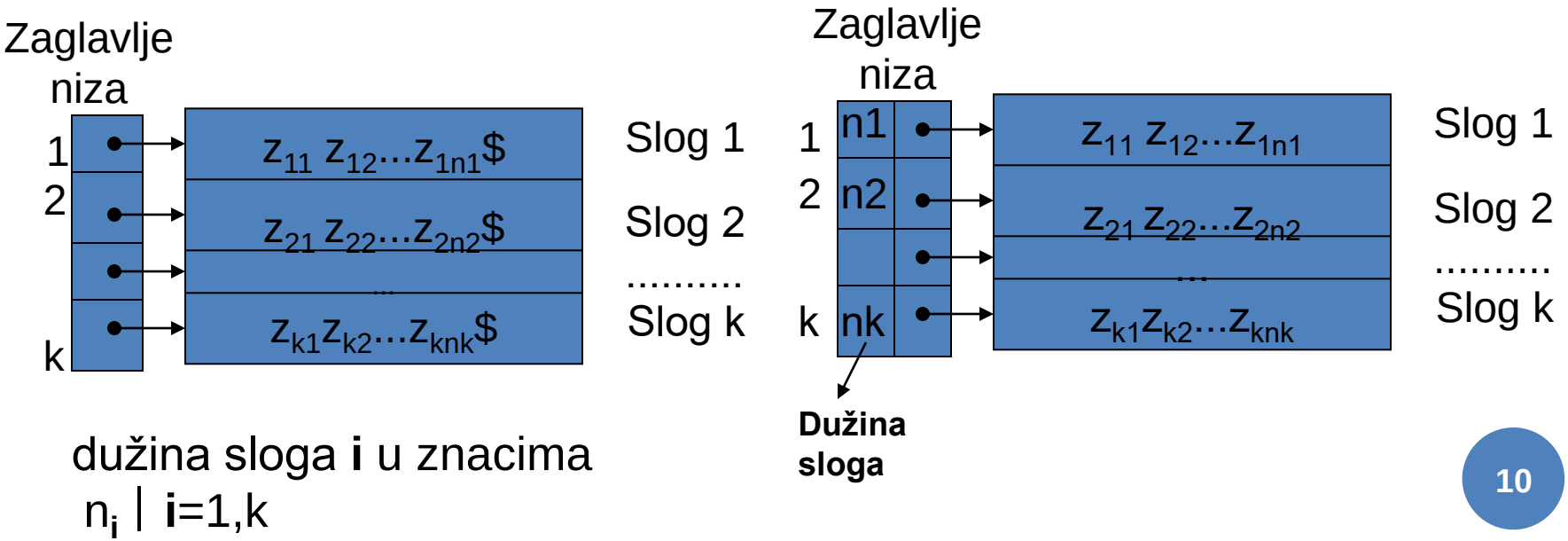




S – ime niza
n – dužina sloga u znacima
k – dužina niza u slogovima

Sekvencijalna reprezentacija niza / 2

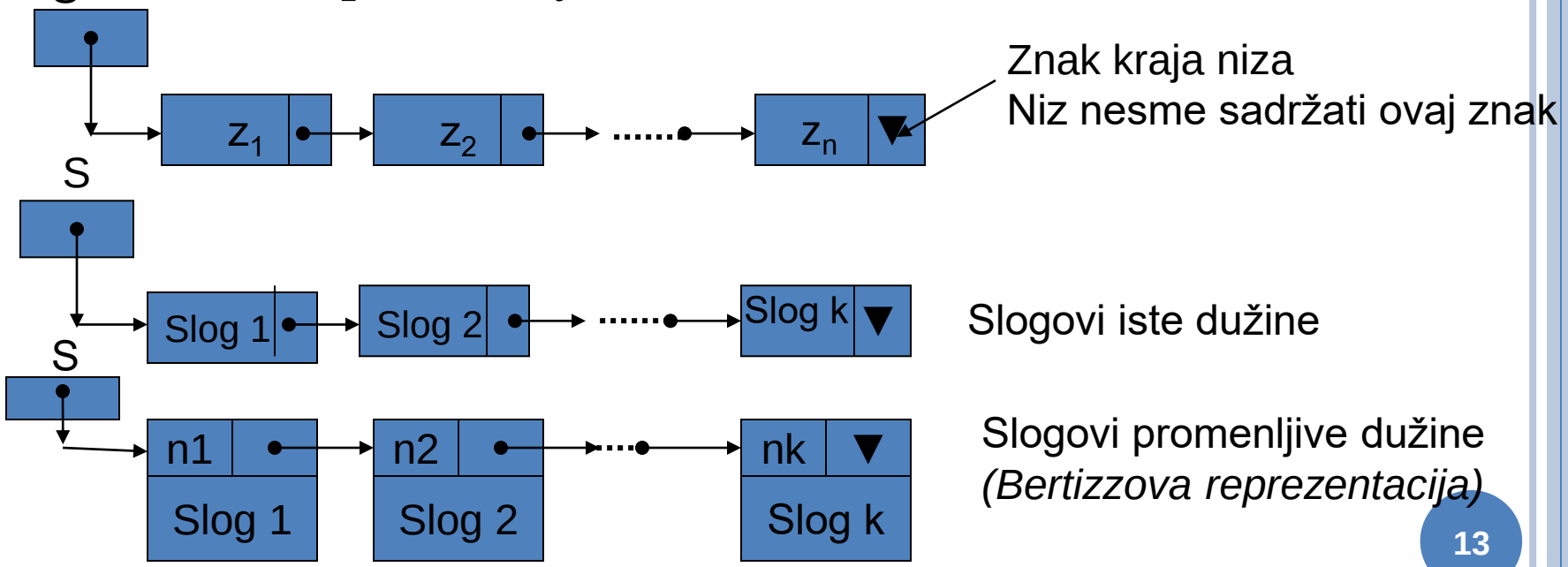
- Slogovi promenljive dužine
  - Slogovi sa kodom kraja ($) – slika levo
  - Slogovi sa memorisanom dužinom sloga – slika desno



dužina sloga **i** u znacima
$n_i \mid i=1,k$

Lančana reprezentacija niza

- Niz se deli u slogove fiksne ili promenljive dužine
- Svaki slog se smešta u poseban memorijski blok
  - Znak po znak
  - Blokovi fiksne veličine
  - Blokovi promenljive veličine



Operacije nad nizovima

- **Osnovne operacije**
  - Izdvajanje podniza (Substring)
  - Indeksiranje (Indexing) ili poklapanje uzoraka (Pattern Matching)
  - Konkatenacija (Concatenation)
  - Nalaženje dužine niza (Length)
- **Kompozitne operacije / obrada reči**
  - Umetanje (Insert)
  - Brisanje (Delete)
  - Zamena (Replace)
- **Algoritmi traženja po tekstu** (Pattern Matching Algorithms)
  - Algoritam grube sile
  - Brzi algoritmi traženja po tekstu
  - Ostali algoritmi

IZDVAJANJE PODNIZA

PS ←SUBSTRING(S,i,n)

S – niz koji obrađujemo
PS – izdvojeni podniz
i – pozicija gde počinje podniz
n – dužina podniza

Funkcija:

- Vraća podniz PS niza S dužine n koji počinje od i-te pozicije

U programskim jezicima:

- FORTAN
- Pascal
- BASIC
- C
- C++
- Java

Primer:

s ← SUBSTRING("Ružica je lepo ime",10,6)
s ← "lepo i" (indeksi 1,2,…)
s ← "epo im" (indeksi 0,1,…)

INDEKSIRANJE

i ← INDEX(T,P)

i – pozicija prve pojave P u T
ili nula (indeksi 1,2,…), odnosno
-1 (indeksi 0,1,…)

T – tekst koji se pretražuje
P – uzorak (pattern) koji se traži

U programskim jezicima:

- FORTAN
- Pascal
- C
- C++
- Java

Funkcija:

- Vraća poziciju gde počinje prva pojava uzorka P u zadatom tekstu T
- Ako se P ne nalazi u T rezultat je 0

INDEKSIRANJE

i ← INDEX(T,P)

Primer 1:

i ← INDEX(“Moj otac i tvoj otac su prijatelji”,”otac”)
i ← 5 (indeksi 1,2,…)
i ← 4 (indeksi 0,1,2,…)

Primer 2:

i ← INDEX(“Moj otac i tvoj otac su prijatelji”,”Moj”)
i ← 1 (indeksi 1,2,…)

Primer 3:

i ← INDEX(“Moj otac i tvoj otac su prijatelji”,”majka”)
i ← 0 (indeksi 1,2,…)
I ← -1 (indeksi 0,1,…)

KONKATENACIJA

S3 ← CONCAT(S1,S2)

Funkcija:

- Vraća niz S3 dobijen konkatenacijom nizova S1 i S2
- Niz S3 sadrži sve znake niza S1 iza kojih slede znaci niza S2

Primer:

S ← CONCAT(“Danilo “,”Kiš”)

S ← “Danilo Kiš”

U programskim jezicima:

- FORTAN
- Pascal
- C
- C++
- Java

Nalaženje dužine niza

n ← LENGTH(S)

Funkcija:

- Vraća dužinu niza S

Primer 1:

n ← LENGTH(“Danilo”)

n ← 6

Primer 2:

n ← LENGTH(“”)

n ← 0

Primer 3:

n ← LENGTH(“ ”)

n ← 1

U programskim jezicima:

- FORTAN
- Pascal
- C
- C++
- Java

Obrada reči

- Ovaj termin se koristi kod obrade ***tekstualnih dokumenata*** (pisama, članaka, izveštaja, itd)
- Operacije:

  Umetanje – novi niz se umeće unutar postojećeg teksta

  Brisanje – iz postojećeg teksta se briše neki niz

  Zamena – u tekstu se jedan niz zamenjuje drugim

- Ovo su ***kompozitne operacije*** nad nizom
  - izvode se pomoću osnovnih operacija

UMETANJE

INSERT(T,k,S)

T – tekst koji se obrađuje

k – pozicija od koje se vrši umetanje

k=0,1,... ili k=1,2,...

S – niz koji se umeće

Funkcija:

- U tekst T se umeće niz S počev od pozicije k

Primer:

T ← INSERT(“ABCD”,2,”xyz”)

T=“AxyzBCD” (indeksi 1,2,…)

T=“ABxyzCD” (indeksi 0,1,…)

UMETANJE

Algoritam N1: insert (T,k,S)

S1 ← substring(T,d,k-d)
S2 ← substring(T,k,length(T)-k+d)
S3 ← concat(S1,S)
T ← concat(S3,S2)
return

Algoritam N2: insert (T,k,S)

T ← concat(concat(substring(T,d,k- d), S),
substring(T,k,length(T)-k+d))
return

Napomena:
d je donja granica indeksa
d=0 ili d=1

Brisanje

DELETE(T,k,n)

T – tekst koji se obrađuje

k – indeks niza koji se briše

n – dužina niza koji se briše

Funkcija:

- Iz teksta *T* se briše niz dužine *n* počev od pozicije *k*
- Izvodi se pomoću prostih operacija

Primer:

DELETE(“ABCDEFGH”,3,4)

T=“ABGH” za d = 1,2,3,…

T=“ABCH” za d = 0,1,2,…

BRISANJE

Algoritam N3: delete (T,k,n)

```
   S1 ← substring(T,d,k-d) // d=0 ili 1
   S2 ← substring(T,k+n,length(T)-k-n+d)
   T ← concat(S1,S2)
return
```

Algoritam N4: delete (T,k,n)

```
// d = 0 ili 1
   T ← concat(substring(T,d,k-d),
         substring(T,k+n,length(T)-k-n+d))
return
```

Primer: Brisanje uzorka P iz T

DeletePat(T,P)

Algoritam N.5: deletePat(T,P)

1. k ← index(T,P) // nalazi indeks prve pojave uzorka
2. T ← delete(T,index(T,P),length(P)) // briše P iz T
3. return

Algoritam N.3

PRIMER: BRISANJE SVIH UZORKA P IZ T

DeleteALL(T,P)

Brisanje svih pojava uzorka P u tekstu T

Algoritam N.5: deleteALL(T,P)

1. k ← index(T,P) // nalazi indeks prve pojave uzorka
2. **repeat while**(k≠0)
3. T ← delete(T,index(T,P),length(P)) // briše P iz T
4. k ← index(T,P) // ažurira indeks
5. **endrepeat**
6. **return**

Algoritam N.3

Složenost: O(n), linearna

Zamena

REPLACE(T,P,Q)

T- tekst koji se obrađuje

P – uzorak koji se menja

Q – uzorak kojim se vrši izmena

Funkcija:

- Zamena prve pojave uzorka P u tekstu T uzorkom Q
- Izvodi se pomoću prostih operacija

Primer:

T ← REPLACE(“ABXYCDXYE”,”XY”,”34”)

T=“AB34CDXYE”

ZAMENA

Algoritam N6: replace (T,P,Q)

k ← index(T,P)

T ← = delete(T,k,length(P))

insert(T,k,Q)

return

Algoritam N.3

Algoritam N.1

Primer: Zamena svih pojava P u T

Zamena svih pojava uzorka P uzorkom Q u tekstu T

Algoritam N.7: replaceAll(T,P,Q)

1. k ← index(T,P) // nalazi indeks prve pojave uzorka
2. **repeat while**(k≠0)
3. T ← replace(T,P,Q) // vrši zamenu P sa Q u T
4. k ← index(T,P) // ažurira indeks
5. **endrepeat**
6. **return**

Algoritam N.6

Složenost: O(n)

Traženje po tekstu

- Traženje po tekstu: pronaći lokaciju zadatog uzorka teksta unutar većeg korpusa teksta (npr. rečenica, pasus, knjiga, itd.).
- Kao i kod većine algoritama, glavna razmatranja za pretraživanje nizova su brzina i efikasnost

- Cilj: Proveriti da li se **uzorak (patern) P nalazi u Tekstu T**
  - Traženje uspešno – uzorak se nalazi u tekstu
  - Traženje neuspešno – uzorak se ne nalazi u tekstu

Traženje po tekstu

- Proveriti da li se **patern P nalazi u Tekstu T**
- **Primena:**
  - Tekst editori
  - Search engines
  - Biološka istraživanja
  - **...**

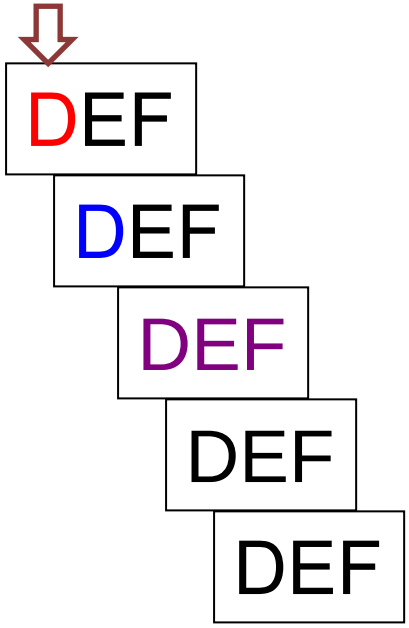
- Algoritmi
  - Traženje po tekstu metodom grube sile: **BruteForceMatching(T,P,INDEX)**
  - **Drugi algoritmi** za brzo traženje po tekstu

Traženje po tekstu metodom grube sile

- **Brute Force** algoritam upoređuje uzorak sa tekstom, **jedan po jedan karakter**, sve dok ne dođe do razlike/nepodudarnosti ili se ne obrade svi karakteri uzorka.
- Ako se svi karakteri uzorka poklapaju sa karakterima iz teksta, uzorak je pronađen.
- Ako je došlo do nepodudardnosti na nekom karakteru, vrši se **pomeranje pretrage** za jedan karakter u tekstu i počinje poređenje sa prvim karakterom uzorka.
- Algoritam se može projektovati tako da se zaustavi na prvom pojavljivanju obrasca, ili po dostizanju kraj teksta

Traženje po tekstu metodom grube sile

- Tekst: ABCDEFGHIJKL
- Patern: DEF
- Metoda:
  - ABDEDEFGH



poredimo **D** i **A** i pošto su različiti napuštamo poređenje i pomeramo patern za jednu poziciju desno

poredimo D i B i pošto su različiti napuštamo poređenje i pomeramo patern za jednu poziciju desno

poredimo D i D, zatim E i E, zatim F i D i pošto su različiti napuštamo poređenje i pomeramo patern za jednu poziciju desno

poredimo D i E i pošto su različiti napuštamo poređenje i pomeramo patern za jednu poziciju desno

poredimo D i D, zatim E i E, zatim F i F – patern je nađen, traženje uspešno

Primer traženja metodom grube sile

*TW*O ROADS DIVERGED IN A YELLOW WOOD *R*OADS
T*W*O ROADS DIVERGED IN A YELLOW WOOD *R*OADS
TW*O* ROADS DIVERGED IN A YELLOW WOOD *R*OADS
TWO ROADS DIVERGED IN A YELLOW WOOD *R*OADS
TWO ***ROADS*** DIVERGED IN A YELLOW WOOD ***ROADS***

Algoritam traženja po tekstu metodom grube sile

Algoritam N.8 Traženje metodom grube sile,
BruteForceMatching(T, P)
// Tekst T i uzorak P su nizovi dužine *n* i *m*, respektivno, a memorisani su kao 1D polja
// Svaki znak je element polja
// Ovaj algoritam nalazi indeks u tekstu T uzorka P, ako se P nalazi u T,
// ili je –1, što pokazuje da se P ne nalazi u T

```
1.  m ← length(P)   // određuje dužinu uzorka
2.  n ← length(T)    // određuje dužinu teksta
3.  k ← 0
4.  while (k ≤ n – m)
5.     //pomeranje uzorka do maks. n-m
6.     j ← 0
7.     while (j < m ∧ T[k + j] = P[j] )
8.        j ← j + 1
9.     if ( j = m)
10.       return  k       //poklapanje od k
11.    else
12.       break while loop //nepoklapanje
13. return  -1     //nema poklapanja u celom T
```

Složenost: $O(n^2)$, kvadratna

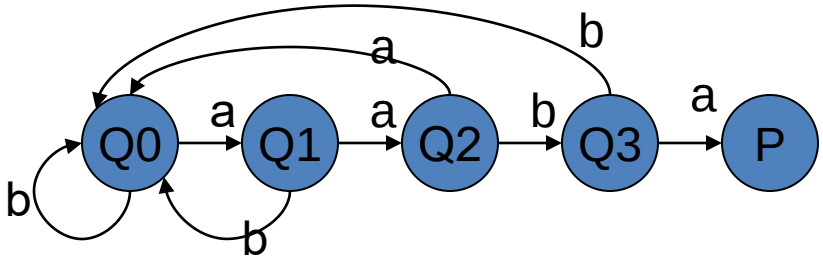
BRZI ALGORITAM TRAŽENJA PO TEKSTU

- Pokušava da reši probleme Brute force pristupa
- Algoritam koristi tablicu kao pomoćnu strukturu podataka
- Tablica se koristi u procesu traženja kako bi se izbeglo upoređivanje karakter po karakter
- Tablica predstavlja graf ili konačni automat koji definiše prelaze iz stanja u stanje, kako „nailaze“ karatkteri iz teksta.
- Tablica se pravi **na osnovu zadatog uzorka P** i ne zavisi od teksta T

Brzi algoritam traženja po tekstu – Primer uzorka i Tablice

Tablica uzorka P=aaba

T / Q	a	b	X
Q0	Q1	Q0	Q0
Q1	Q2	Q0	Q0
Q2	Q0	Q3	Q0
Q3	P	Q0	Q0



Graf uzorka

Azbuka uzorka Σ=(a,b)
X – bilo koji znak koji ne pripada Σ

Qi – svi mogući podnizovi uzorka

Q0= prazan niz
Q1 = a
Q2 = aa
Q3 = aab
Q4 = aaba (=P)

Brzi algoritam traženja po tekstu

Algoritam N.9. Brzo traženje po tekstu
PatternMatching(T,P,F,INDEX)
// Tekst T, uzorak P i tablica F(Q,T) su u memoriji
// Tekst T je memorisan kao 1D polje, svaki znak je element polja
// Ovaj algoritam nalazi indeks INDEX uzorka P u tekstu T, ako se
// P nalazi u T, ili je INDEX=0, što pokazuje da se P ne nalazi u T

1. lt ← length(T)
2. k ← 1
3. Q ← "" // prazan niz Q0
4. **while**(Q ≠ P and k ≤ lt)
5. Q ← F(Q,T[k]) // nalazi sledece stanje
6. k ← k + 1
7. **end while**
8. **if** (Q = P)
9. **then** INDEX ← k – length (P) // traženje uspešno
10. **else** INDEX ← 0 // traženje neuspešno
11. **exit**

- Algoritam N.9 koristi tablicu F kao pomoćnu strukturu podataka
- Tablica F se pravi na osnovu zadatog uzorka P i ne zavisi od teksta T

Složenost algoritma: O(lt), linearna

KNUTH-MORRIS-PRATT (KMP) ALGORITAM

- KMP algoritam se razlikuje od Brute-force algoritma po tome što prati informacije dobijene iz prethodnih poređenja.
- Izračunava se funkcija greške (**f**) koja pokazuje kako da se veći deo poslednjeg poređenja može ponovo iskoristiti.
- Konkretno, **f** je definisana kao

  najduži **prefiks** uzorka **P[0,..,j]**

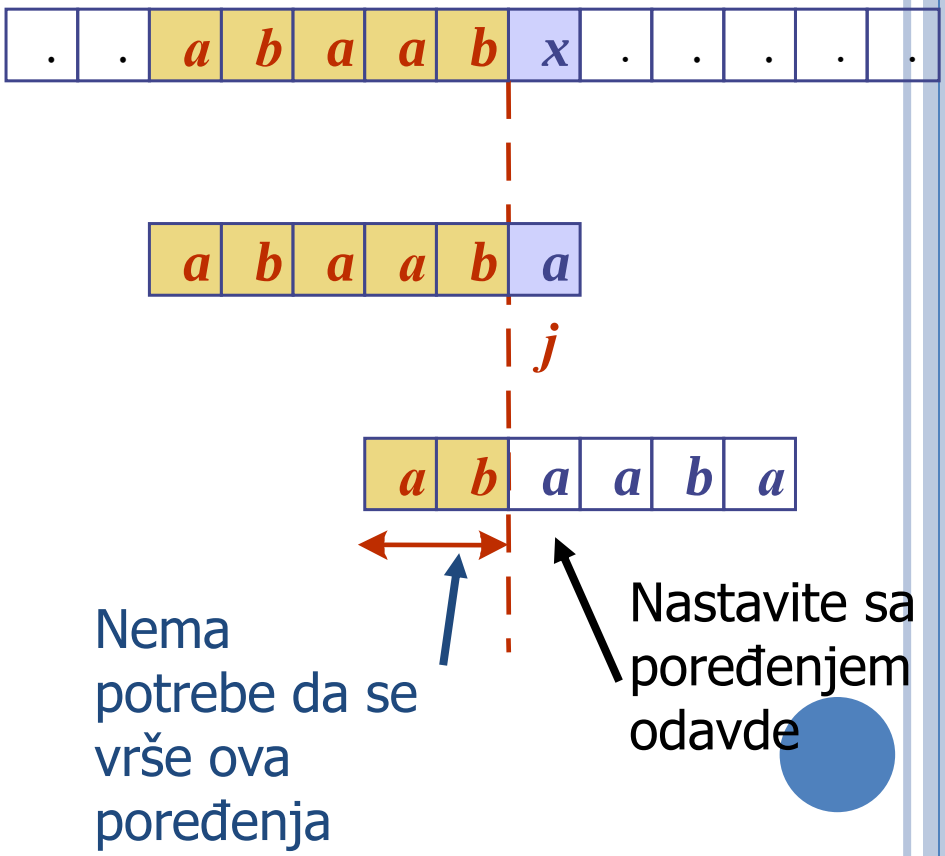
  koji je takođe **sufiks** od **P[1,..,j]**

  *Napomena: **nije** sufiks od P[0,..,j]

Kako radi KMP Algoritam

- Ako razmotrimo poređenje uzorka sa tekstom kao kod Brute-force algoritma,
- kada se desi nepoklapanje, koliko **najviše** možemo da pomerimo uzorak da bi izbegli redundantna poređenja?

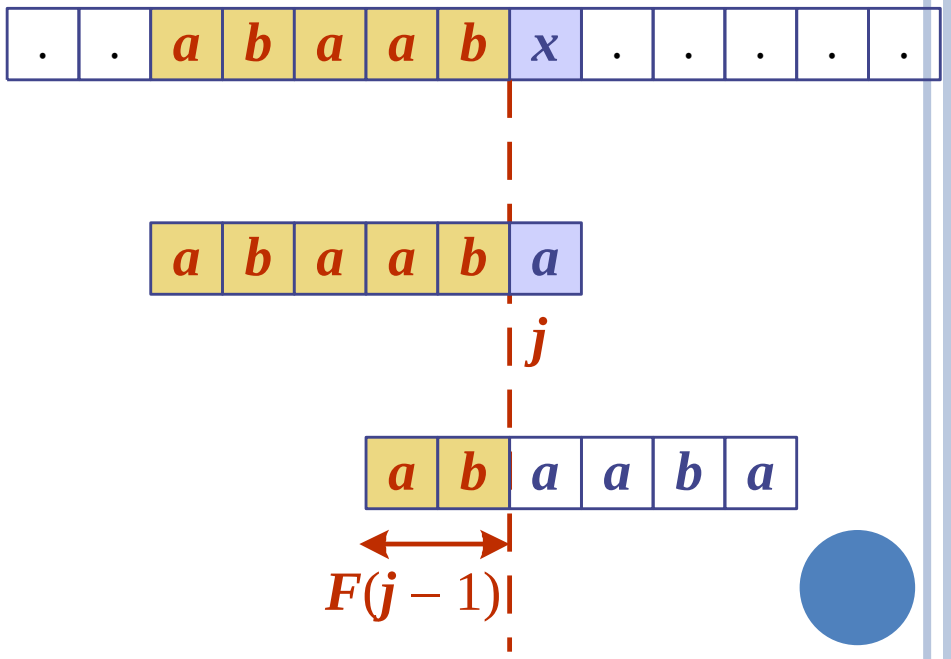
- Odgovor:
  najduži prefiks uzorka ***P***[0..***j***] koji je sufiks ***P***[1..***j***]



KMP Funkcija greške

- Knuth-Morris-Pratt algoritam vrši predobradu uzorka da nađe poklapanja prefiksa uzorka sa samim uzorkom
- **Funkcija greške** $F(j)$ se definiše kao dužina najdužeg prefiksa u $P[0..j]$ koji je takođe sufiks u $P[1..j]$
- KMP algoritam modifikuje Brute-force algoritam tako da ako se desi nepoklapanje na $P[j] \neq T[i]$ i $j > 0$, dodeljujemo $j \leftarrow F(j - 1)$

j	0	1	2	3	4	5
$P[j]$	*a*	*b*	*a*	*a*	*b*	*a*
$F(j)$	0	0	1	1	2	3



Pseudokod KMP Algoritma

Algoritam N.10. KMP algoritam

***KMPMatch*(*T*, *P*)**

```
1.  F ← failureFunction (P)
2.  i ← 0
3.  j ← 0
4.  while (i < n)
5.      if T[i] = P[j]
6.          if j = m – 1
7.              return i – j  // poklapanje
8.          else
9.              i ← i + 1
10.             j ← j + 1
11.     else
12.         if j > 0
13.             j ← F[j - 1]
14.         else
15.             i ← i + 1
16. return -1 // nema poklapanja
```

- Funkcija greške može da se implementira kao 1D polje i može da se izračuna za $O(m)$
- U svakoj iteraciji tj while petlji,
  - i se povećava za 1, ili
  - $i - j$ se povećava za najmanje 1 (zapazite da je $F(j-1) < j$)
- Dakle, ne postoji više od $2n$ iteracija u while petlji
- Zaključak, KMP's algoritam se izvršava za $O(m+n)$

Izračunavanje Funkcije greške za KMP

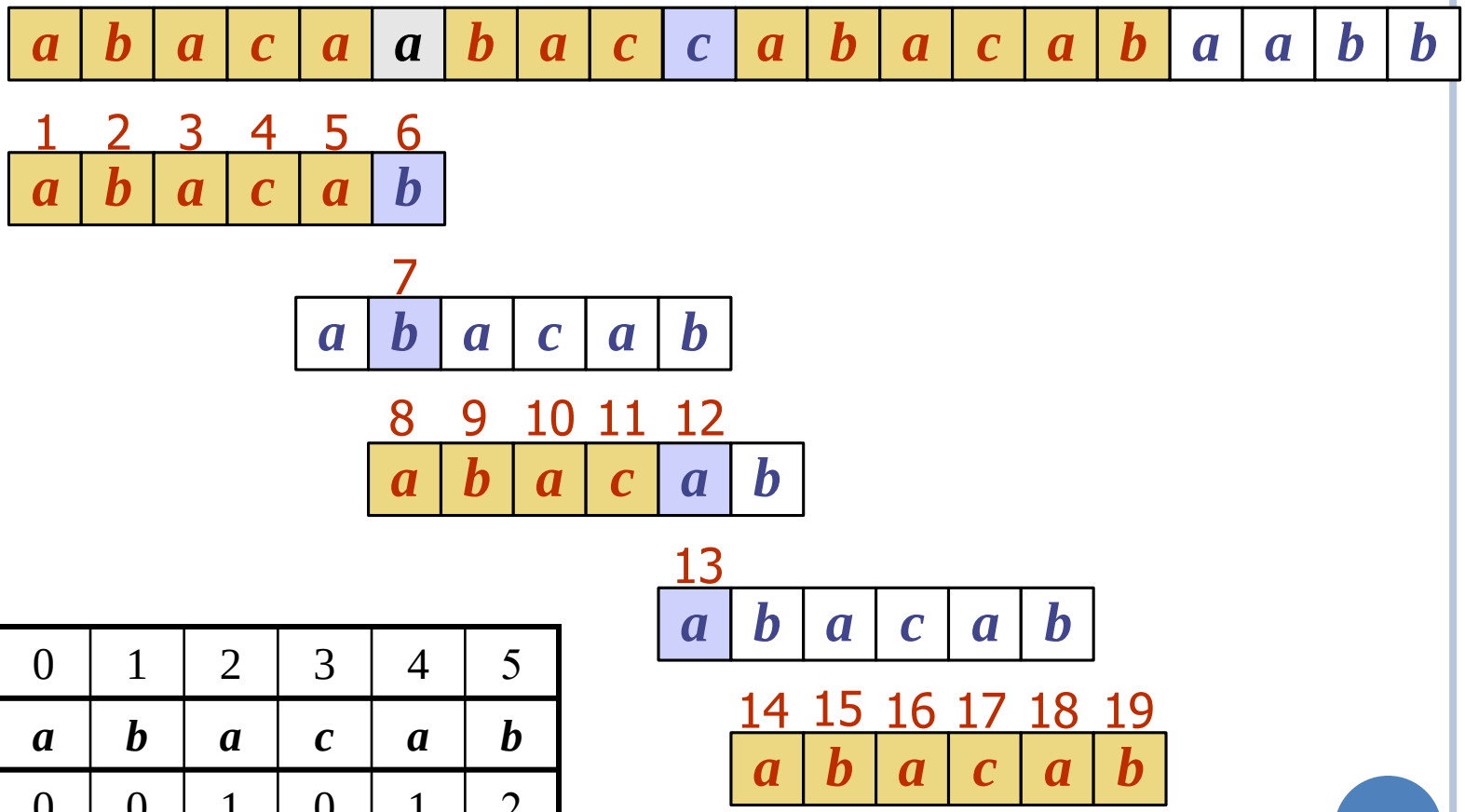
Algoritam N.11 Funkcija greške za KMP

***failureFunction*(*P*)**

```
F[0] ← 0
i ← 1
j ← 0
while i < m
    if P[i] = P[j]
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j - 1]
    else
        F[i] ← 0 { no match }
        i ← i + 1
```

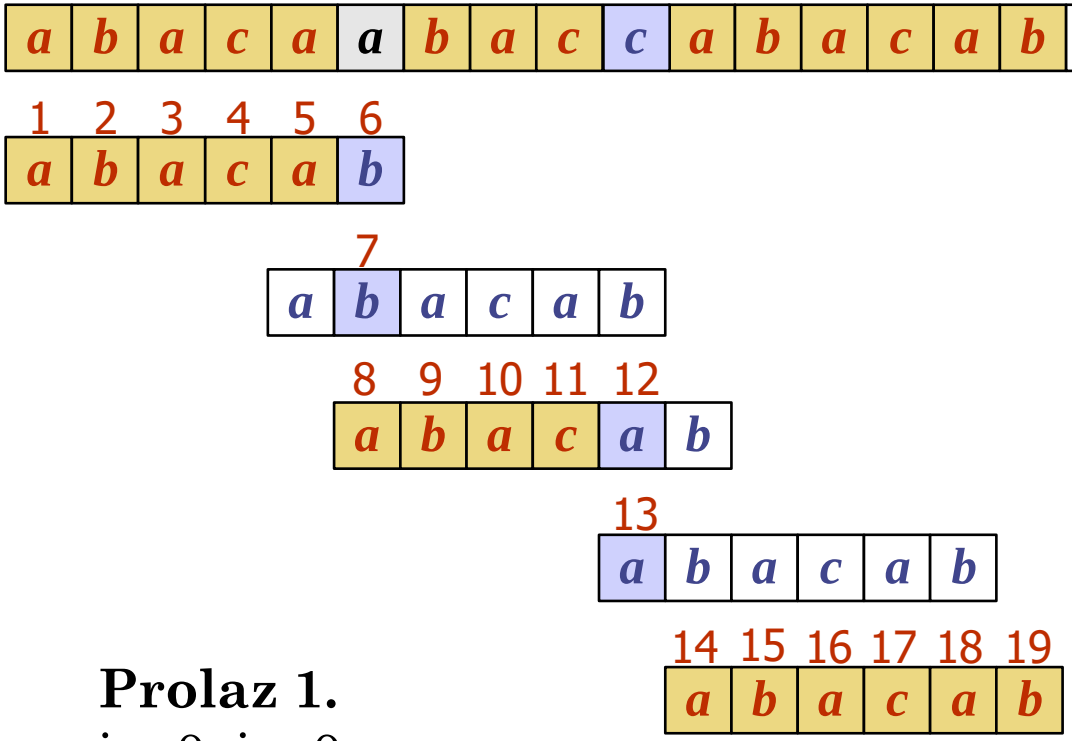
Složenost reda $O(m)$

Primer rada KMP algoritma



j	0	1	2	3	4	5
***P*[*j*]**	***a***	***b***	***a***	***c***	***a***	***b***
***F*(*j*)**	0	0	1	0	1	2

Primer rada KMP algoritma



Prolaz 1.

i = 0, j = 0

i =i+1, j = j+1,… **i=j=5 / nepoklapanje**

j = F[j–1] = F[4] = 1

Prolaz 2. …

```
KMPMatch(T, P)
1.  F ← failureFunction (P)
2.  i ← 0
3.  j ← 0
4.  while (i < n)
5.      if T[i] = P[j]
6.          if j = m – 1
7.              return i – j  // poklapanje
8.          else
9.              i ← i + 1
10.             j ← j + 1
11.     else
12.         if j > 0
13.             j ← F[j - 1]
14.         else
15.             i ← i + 1
16. return -1 // nema poklapanja
```

j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
F(*j*)	0	0	1	0	1	2

Rabin Karp algoritam

- Algoritam izračunava hash vrednost za uzorak i za podniz od M-karaktera sa kojim upoređuje uzorak.
- Ako su **heš vrednosti različite**, algoritam uzima novih M karaktera, od pozicije +1 i izračunava heš funkciju sekvence.
- Ako su **heš vrednosti jednake**, algoritam će primenom Brute Force izvršiti poređenje karakter po karakter uzorka i sekvence od M karaktera.
- Na ovaj način postoji samo jedno poređenje (ako nema poklapanja heš vrednosti).
- Brute Force je potrebna samo kada se hash vrednosti podudaraju.

Primer rada Rabin Karp algoritma

- Hash vrednost: za “AAAAA” je 37, za “AAAAH” je 100

1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAH 37≠100 (**1 poređenje**)

2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAH 37≠100 (**1 poređenje**)

3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAH 37≠100 (**1 poređenje**)

4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAH 37≠100 (**1 poređenje**) ...

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAH

100=100 (**6 poređenja**)

Rabin Karp algoritam

Algoritam N.12. *Rabin Karp*

RabinKarp (T, P)

// Uzorak je dužine M karaktera

1. *hash_p* ← **HashValue** (P, M) // hash vrednost uzorka
2. *hash_t* ← **HashValue** (T, M) // hash vrednost prvih M karaktera u telu teksta
3. **repeat**
4. **if** (*hash_p* = *hash_t*)
5. // Brute force poređenje uzorka i selektovane sekcije teksta
6. *hash_t* ← **HashValue**(*sledeća sekcija teksta, pomeraj za jedan karakter*)
7. **until** (*kraj teksta* **or** *brute force poređenje* = **true**)

Rabin Karp – uobičajena pitanja

- Koja se heš funkcija koristi za izračunavanje vrednosti za sekvence znakova?
- Zar nije potrebno mnogo vremena za heširanje svake sekvence od M-karaktera u telu teksta?

Algoritmi traženja po tekstu

Algoritmi sa jednim uzorkom (Single pattern algorithms)

	Preprocessing time	**Matching time**
Naïve string search algorithm	0 (no preprocessing)	Θ(n m)
Rabin-Karp string search algorithm	Θ(m)	average Θ(n+m), worst Θ(n m)
Finite automaton	O(m |Σ|)	Θ(n)
Knuth-Morris-Pratt algorithm (KMP)	Θ(m)	Θ(n)
Boyer-Moore string search algorithm (BM)	Θ(m)	average Θ(n/m), worst Θ(n)
Bitap algorithm	Θ(m+|Σ|)	Θ(n)
Baeza-Yates and Gonnet string search algorithm		

Σ je azbuka, *m* je dužina uzorka, a *n* dužina teksta koji se pretražuje

Pitanja, ideje, komentari