IAL – 10. přednáška

Vyhledávání v textu

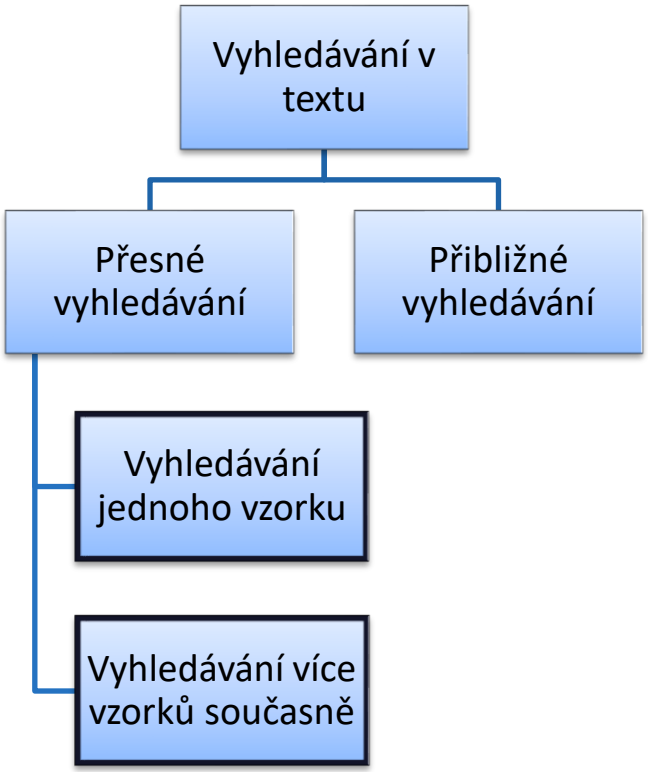
19. a 20. listopadu 2024

Obsah přednášky

- Vyhledávání jednoho vzorku
  - Klasický (naivní) algoritmus
  - Knuth-Morris-Prattův algoritmus
  - Boyer-Mooreův algoritmus
  - Rabin-Karpův algoritmus
- Vyhledávání více vzorků
  - Písmenkové stromy
    - Komprimovaná trie
  - Algoritmus Aho-Corasicková
- Sufixové stromy

Vyhledávání v textu

- Důležitá skupiny algoritmů pro práci s textem.
- Vyhledáváme přesný výskyt vzorku (jehly, podřetězce) ve větším textu.
- Budeme používat značení:
  - Vyhledávaný vzorek (pattern): `p`
    - i-tý znak vzorku: `p[i]`
    - délka vzorku: `m` nebo `pl`
  - Prohledávaný text: `t`
    - i-tý znak prohledávaného textu: `t[i]`
    - délka prohledávaného textu: `n` nebo `tl`



Klasický algoritmus

- *Naivní* algoritmus, *brute-force* algoritmus
- Přikládá vzorek k textu zleva doprava.
- Porovnává symboly textu a vzorku zleva doprava.
- Při neshodě symbolů:
  - Posune vzorek o jednu pozici doprava.
  - Porovnává symboly zleva doprava, od prvního symbolu vzorku a odpovídajícího symbolu v textu.

- ***Pozn:*** Algoritmus vrací pozici prvního výskytu hledaného vzorku v textu. Pokud se vzorek v textu nevyskytuje, vrací pozici *za textem*.

Klasický algoritmus

```
int function Match (char *t, char *p, int pl, int tl)
// vrací index prvního výskytu, při neúspěchu vrátí hodnotu TL
  auxStartT ← 0                          // inicializace
  posT ← 0
  posP ← 0
  while posT < tl and posP < pl:
    if t[posT] = p[posP]:              // posun po vzorku v řetězci
      posT ← posT + 1
      posP ← posP + 1
    else:                    // posun zač. řetězce a nové porovnání
      auxStartT ← auxStartT + 1
      posT ← auxStartT
      posP ← 0
  if posP = pl:
    return auxStartT          // našel
  else:
    return posT              // nenašel a vrátil hodnotu TL
```

Analýza klasického algoritmu

- Nejlepší případ:
  - Vzorek se vyskytuje hned na počátku řetězce, provede se **`pl`** porovnání.
- Nejhorší případ:
  - Na každé startovací pozici dojde k (`pl-1`) shodám. Pak se provede **`mn`** srovnání a algoritmus má složitost ***O(mn).***
    (příklad: `P='AAA...AB'` a `T='AAA...AAA'`).
- Přirozené jazyky:
  - Nejhorší případ je zde neobvyklý.
  - Statistiky ukazují cca 1,1 porovnání na jeden znak řetězce `t`.
- Algoritmus vyžaduje návraty v textu!
  - Pro některé aplikace nepřijatelné.
  - `posT ← auxStartT` v cyklu.
- ***Pozn:*** Není-li `p[0]` v řetězci `t` obsažen, provede se **`tl`** srovnání.

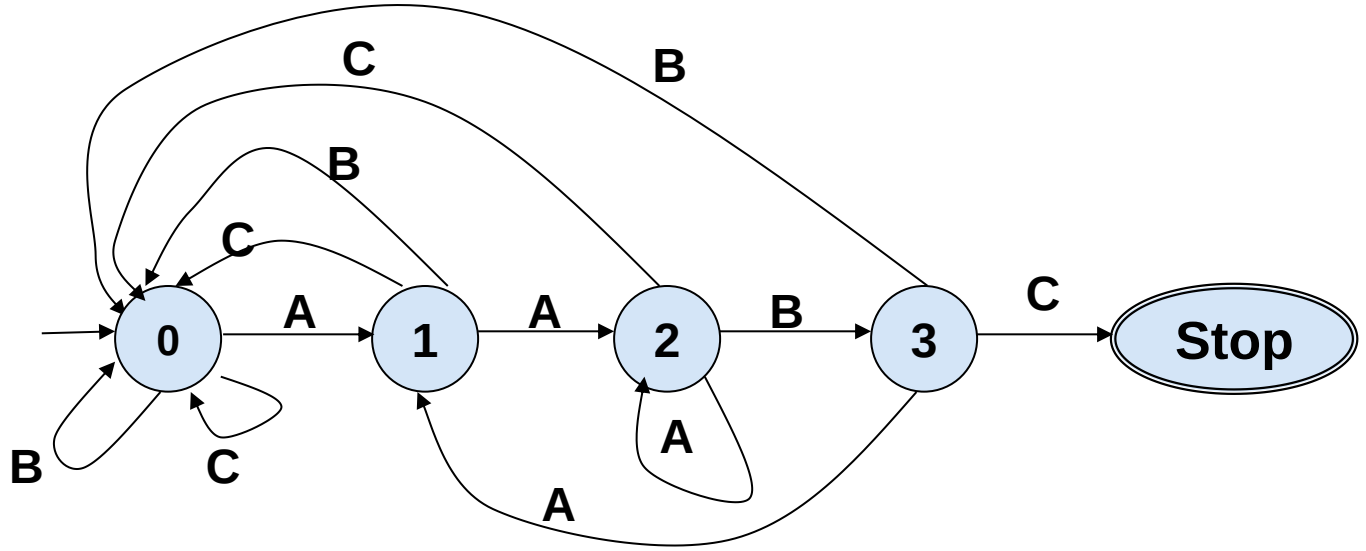
Knuth-Morris-Prattův algoritmus (KMP)

- Využívá *princip konečného automatu*.
- Přikládá vzorek k textu zleva doprava.
- Porovnává symboly textu a vzorku zleva doprava.
- Při neshodě symbolů:
  - Nevrací se v textu zpět, ale vyzkouší další možné přiložení vzorku, které odpovídá přečtené části textu.
  - Symbol textu, na kterém došlo k neshodě, porovná s jiným *vhodným* symbolem vzorku.

Text:	clanekokokosu
Aktuální přiložení vzorku:	kokos
Další možné přiložení:	kokos

Princip konečného automatu

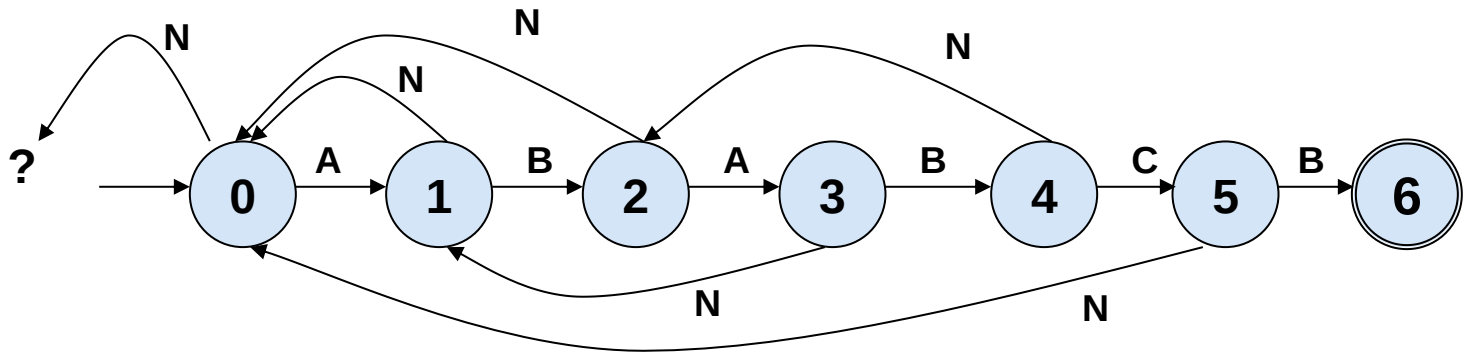
- Nechť Σ je abeceda a o je kardinalita abecedy Σ. Pak z každého uzlu vychází o orientovaných hran, oceněných jednotlivými znaky abecedy.
- Pro vzorek `AABC` a abecedu `{A,B,C}` dostaneme automat:



- Nevýhoda: z každého uzlu vychází tolik hran, kolik je znaků abecedy.

KMP – vyhledávací automat

- Vyhledávací automat používá dva typy hran:
  - Dopředné hrany:
    - Označeny symboly vzorku.
    - Použijí se, pokud se v textu nachází daný symbol.
  - Zpětné hrany:
    - Použijí se, pokud se v textu nachází jiný symbol.
    - Po použití zpětné hrany se nečte nový symbol, ale provede se další krok se stejným symbolem.
    - Je-li potřeba jít zpět ze stavu 0, je načten nový znak z textu.
- Pro vzorek `ABABCB` má KMP automat tvar:



KMP – vyhledávací automat

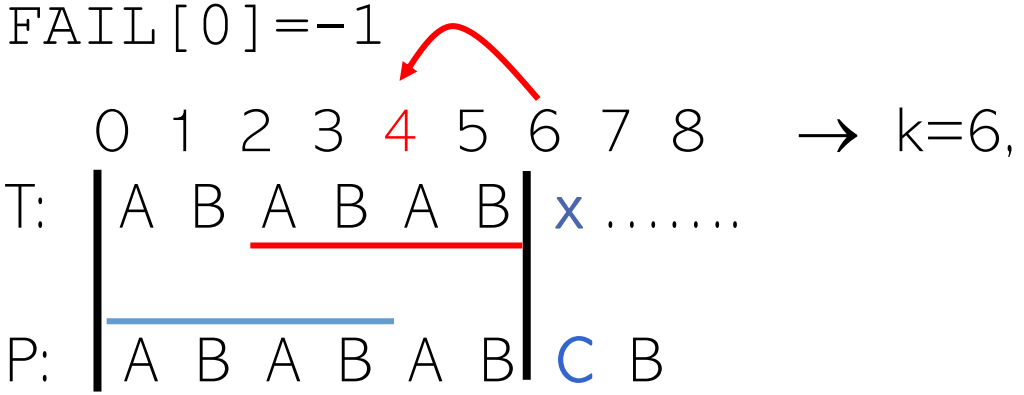
- Reprezentace KMP automatu:
  - Vzorek P – udává označení dopředných hran.
  - Vektor FAIL – udává cílový stav zpětných hran.
    - Obsahuje prvky typu `int` a jeho velikost odpovídá délce vzorku.
    - `FAIL[0]=-1` reprezentuje čtení nového znaku v textu.
    - *Pozn.:* Pro vyhledání všech výskytů daného slova by vektor FAIL měl velikost délka vzorku + 1.
- Jak určit cílový stav?
  - Potřebujeme najít další možné přiložení vzorku a žádné nevynechat.
  - Hledáme nejdelší možný vlastní prefix vzorku, který odpovídá sufixu, který jsme úspěšně přečetli.

$$FAIL[k] = \max r\{(r < k)\ and\ (P_0 \dots P_{r-1}) = (P_{k-r} \dots P_{k-1})\}$$

KMP – vektor FAIL

Příklad tvorby vektoru pro vzorek `P=ABABABCB`

`FAIL[0]=-1`



Je-li **x<>C,** pak další možné místo, na kterém může vzorek v textu začínat, je třetí pozice, protože došlo k nesouhlasu po přečtení prefixu délky 6 (nesoulad na indexu 6) a protože platí:
(P0...P3) = (P2...P5) .
Nové porovnání může začít ve stavu **4** (protože víme, že symboly 0..3 se v textu nacházejí) a tedy `FAIL[6]=4` .
Platí tedy: **`FAIL: -1 0 0 1 2 3 4 0`**

KMP – vektor FAIL

```
procedure KMPFindFail (char *p, int pl, int fail[pl])
  // varianta pro vyhledávání jednoho výskytu vzorku
  fail[0] ← -1
  for k ← (1, PL-1):
    r ← fail[k-1]
    while (r ≥ 0) and (P[r] ≠ P[k-1]):
            // použit zkratový booleovský výraz!
      r ← fail[r]
    fail[k] ← r + 1
```

- Celkový počet porovnání je (*2m-3*). To představuje *lineární* časovou složitost.
- *Pozn.:* Pro variantu vyhledávání všech výskytů vzorku, bychom cyklus `for` provedli až do `PL`.

KMP – algoritmus

```
int KMPMatch(char *t, char *p, int pl, int tl, int fail[pl])
  posT ← 0
  posP ← 0
  while (posT < tl and posP < pl):
    if posP < 0:     // žádná shoda, posun v textu dopředu
      posP ← 0
      posT ← posT + 1
    else:
      if (t[posT] = p[posP]):    // shody, inkrementace
        posT ← posT + 1
        posP ← posP + 1
      else:                      // neshoda, zpětná hrana
        posP ← fail[posP]
  if posP = pl:
    return posT - pl         // našel, vrací začátek vzorku
  else:
    return posT              // nenašel, vrací hodnotu TL
```

KMP - zhodnocení

- Konstrukce automatu: *O(m)*
- Vyhledávání – maximálně 2n porovnání: *O(n)*
- Celkově: ***O(n+m)***

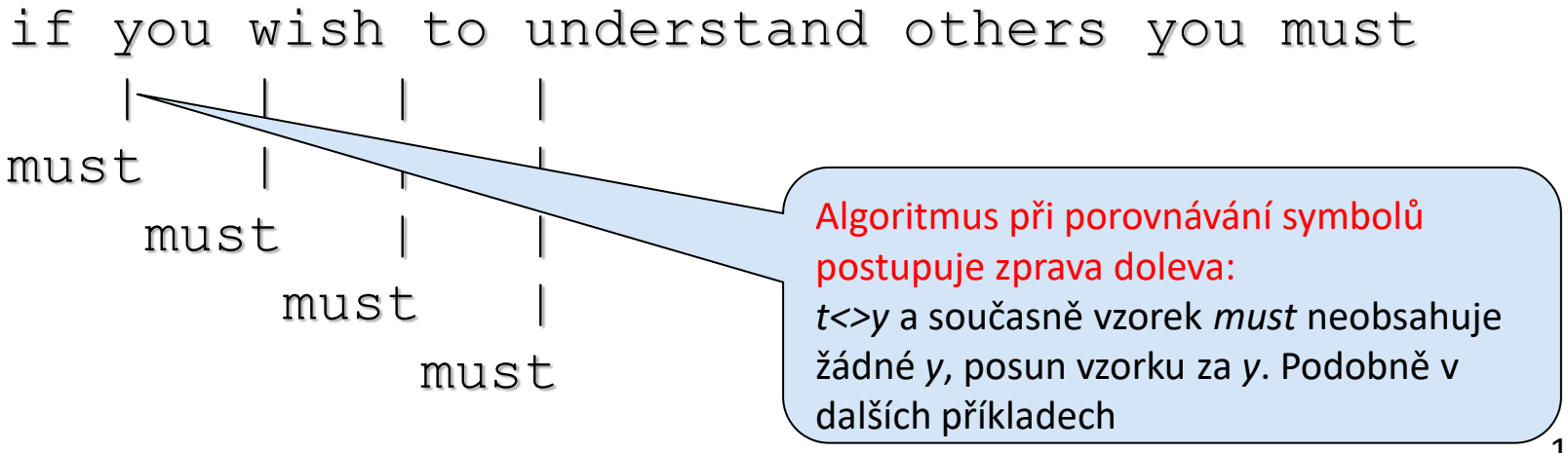
- Přirozené jazyky:
  - Některé empirické studie ukazují, že KMP algoritmus i naivní algoritmus provedou přibližně stejný počet porovnání.
  - KMP nejde v textu zpět.

Boyer-Mooreův algoritmus

- Pokouší se o větší skoky v textu.
- Přikládá vzorek k textu zleva doprava.
- Porovnává symboly textu a vzorku **zprava doleva**:
  - Díky tomu nemusí být některé symboly textu vůbec porovnány se symboly vzorku (lze je přeskočit).
- Při neshodě symbolů využívá dvě pravidla:
  - První je odvozeno od nejpravějšího výskytu symbolu z textu ve vzorku.
  - Druhé je odvozeno od opakujících se podřetězců ve vzorku.
  - *Pozn.:* Čím je vzorek delší, tím větší počet znaků můžeme obvykle přeskočit.

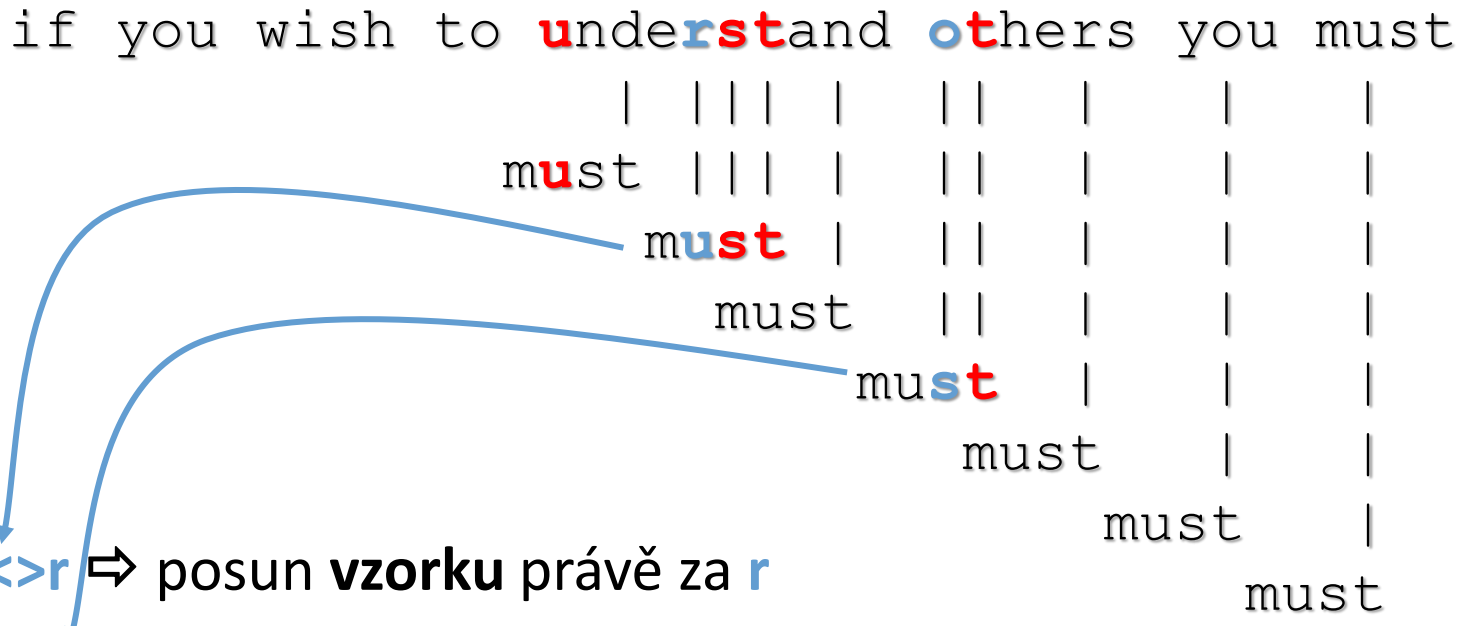
Boyer-Mooreův algoritmus

- Narazíme-li v textu na znak, který se ve vzorku vůbec nevyskytuje, můžeme vzorek posunout až za tuto pozici v texu.
  - Nalezli jsme znaky, které se nemohou rovnat a můžeme je přímo přeskočit.



Boyer-Mooreův algoritmus

- Dále porovnání vypadá takto:



u<>r ⇨ posun **vzorku** právě za **r**

s<>o ⇨ posun vzorku právě za **o**

Provede se jen 18 porovnání pro nalezení vzorku na indexu 37 (indexováno od 0).

BM algoritmus – 1. pravidlo

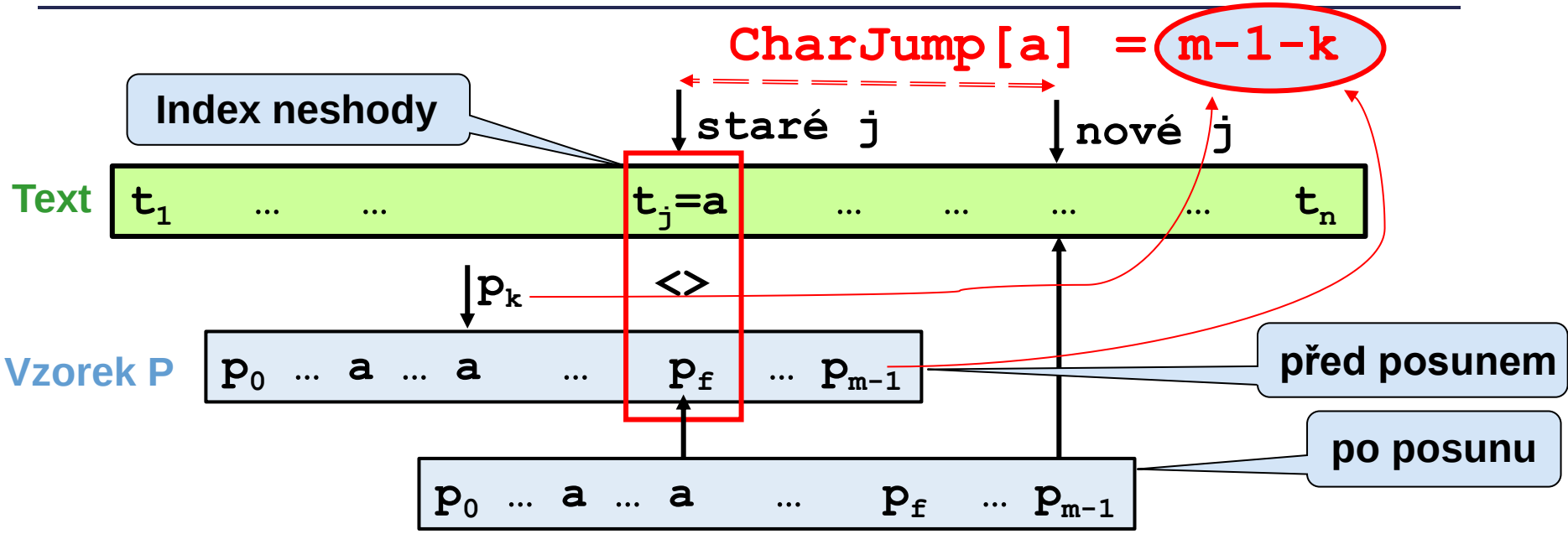
- ***Bad character rule:***
  - Odvozena od symbolu, který se nachází v textu a nesouhlasí se symbolem vzorku. Závisí na znaku v textu – $\mathbf{t_j}$.
  - Určuje počet pozic, o které lze při nesouhlasu porovnávaného vzorku *skočit* dopředu.
- Pro různé symboly abecedy – různě velké skoky:
  - Lze je uložit do pole **`CharJump`**, které bude indexováno typem *znak* (bude mít počet prvků shodný s počtem prvků použité abecedy).
  - K neshodě může dojít pro libovolnou pozici ve vzorku - pro řízení prohlížecího algoritmu je pohodlnější uchovávat hodnotu, o kterou se má zvýšit index **`j`** (index v textu), od něhož se zahájí testování ve směru zprava-doleva, než počet pozic, o který se vzorek posouvá podél prohledávaného textu.

BM algoritmus – 1. pravidlo

- **Délka skoku** závisí na tom, kde ve vzorku se nachází symbol z textu, pro který došlo k neshodě:
  - pokud se t_j vůbec nevyskytuje ve vzorku **P**, lze poskočit o m pozic.
  - v případě, že se t_j ve vzorku nachází, je potřeba provést nejmenší možný skok – odvozený od nejpravějšího výskytu znaku ve vzorku.

- *Pozn.:* Pokud se symbol t_j nachází ve vzorku vpravo od neshody, pak použití tohoto pravidla by vedlo na návrat vzorku podél textu zpět. To ale nedovolí druhé pravidlo. Pokud není implementováno, je třeba toto ošetřit.

BM algoritmus – 1. pravidlo



- Podle symbolu v textu na indexu neshody se určí hodnota, o kterou se má zvýšit index j.
- Pro délku posunu je rozhodující, kde ve vzorku nejvíce vpravo se nachází daný symbol z textu (písmeno `a`). Pokud se nachází na indexu `k`, pak můžeme skočit vpřed o `(m-1-k)` pozic.

Pozn.: `m` je délka vzorku.

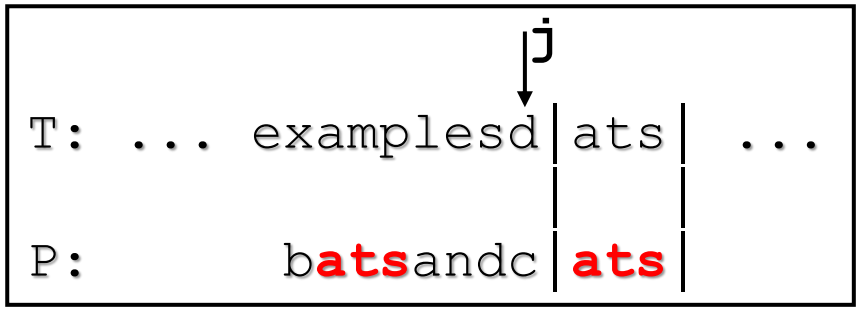
BM algoritmus – 1. pravidlo

```
procedure ComputeJumps (char *p, int CharJump[cardABC])
// stanovení hodnot pole CharJump, určující posuv vzorku
  for i ← (0, cardABC-1):
    ch ← char(i)          // pole bude indexováno znakem
    CharJump[ch] ← length(p)
  for k ← (0, length(p)-1):
    CharJump[p[k]] ← length(p)-1-k
                  // viz (m-1-k) na předch. snímku
```

- ***Pozn:*** V této variantě funguje pouze s využitím obou pravidel.

BM algoritmus – 2. pravidlo

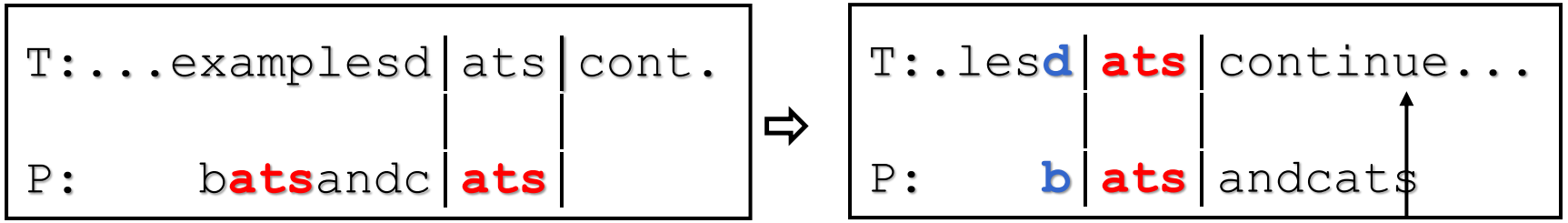
- ***Good suffix rule:***
  - Využívá opakující se podřetězce v řetězci.
  - Pokud úspěšně porovnáme několik symbolů vzorku a textu a potom narazíme na neshodu, potom další smysluplné přiložení vzorku k textu je takové, které k přečtenému sufixu přiloží další nejpravější výskyt tohoto podřetězce ve vzorku.
  - Navíc se bere v úvahu symbol, který předchází danému podřetězci – ten musí být jiný, než při neshodě, jinak by ani toto přiložení nemohlo uspět.



- *Pozn.:* Kombinace `ats` se ve vzorku vyskytuje dvakrát – udává další možné přiložení vzorku.

BM algoritmus – 2. pravidlo

- Pro každou pozici ve vzorku, potřebujeme určit, jak se změní nové `j` (index do textu).
- Lze realizovat polem **`MatchJump`**, jehož velikost odpovídá délce vzorku.
- Pro každou pozici `k` ve vzorku, potřebujeme najít nejpravější index `r`, pro který platí:
  $(p_r \; .. \; p_{r+m-k-2}) = (p_{k+1} \; .. \; p_{m-1})$ a současně $p_{r-1} <> p_k$
  pak: **`MatchJump[k]= m-r`**,
  kde `m` je délka vzorku.

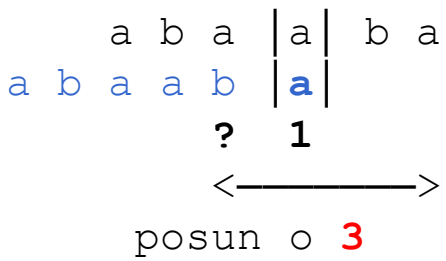


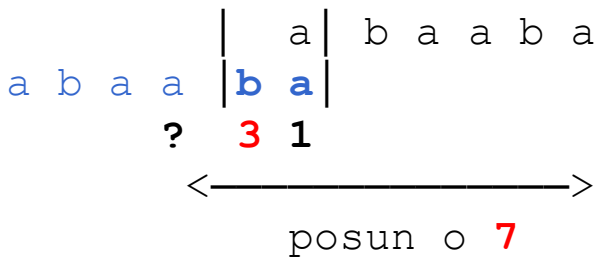
BM algoritmus – 2. pravidlo

- Jak ale určíme hodnotu pole **`MatchJump`**, jestliže už ve vzorku nenajdeme další výskyt celého právě porovnaného sufixu, kterému navíc předchází jiný symbol?
- Použijeme nejdelší možný prefix, který se shoduje s částí přečteného sufixu:
  pak: **`MatchJump[k]= m-(k+1)+m-q = 2m-k-1-q,`**
  - `m` je délka vzorku,
  - `k` je index neshody (pro indexování od 0),
  - `q` je délka nejdelšího prefixu shodného se sufixem.

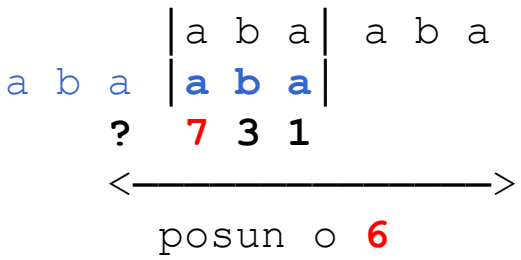
- *Pozn. 1*: Celkově lze hodnoty pole `MatchJump` určit takto:
  `MatchJump[k]= 2m-r-k-1-q`
- *Pozn. 2*: **`MatchJump[m-1] = 1`**

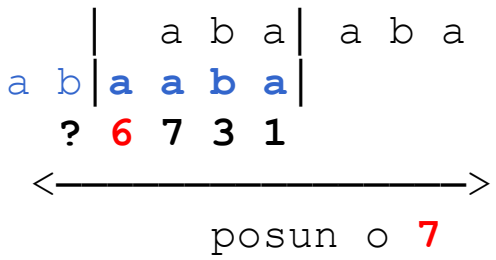
Příklad: výpočet MatchJump pro vzorek **abaaba**,
Ve třetím řádku je vznikající vektor **MatchJump**. Nová hodnota je červená.
Nad otazníkem je neshoda.

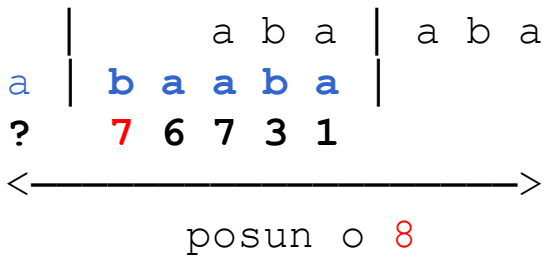




Všimněme si, že první **ba** a druhé **ba** ve druhém kroku není použito, protože obě předchází **a** a nedochází tedy k **nesouhlasu** na pozici před příponou. Dojde-li k nesouhlasu na 4. pozici vzorku, neexistuje žádná poloha pro zarovnání s jiným **a** vzorku, než s prvním a posun je o 7.







Výsledkem pro řetězec P: a b a a b a
je pole MatchJump: 8 7 6 7 3 1

BM algoritmus

```
int BMA (char *p, char *t, int CharJump[cardABC],
             int MatchJump[lengthP])
  // funkce vrací index prvního výskytu vzorku v daném textu
  posT ← length(p) - 1
  posP ← length(p) - 1
  while posT < length(t) and posP ≥ 0:
    if t[posT] = p[posP]:
      posT ← posT - 1
      posP ← posP - 1
    else:
      posT ← posT +
                max(CharJump[t[posT]], MatchJump[posP])
      posP ← length(p)-1
  if posP < 0:
    return posT + 1              // shoda - vrací index
  else
    return length(t)             // shoda se nenašla
```

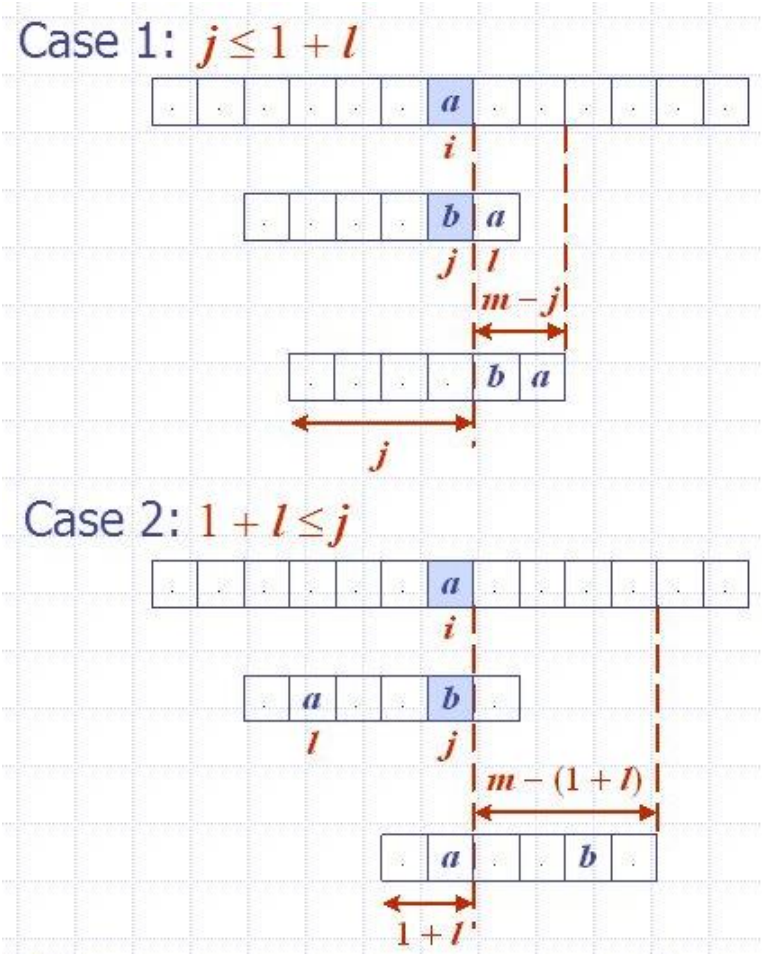
BM algoritmus – zhodnocení

- Chování BMA závisí na kardinalitě abecedy a na opakování podřetězců ve vzorku.
- Nejhorší případ:
  - Pokud se vzorek v textu nevyskytuje: *O(n+m)*
  - Pokud se vzorek v textu vyskytuje a hledáme všechny výskyty: *O(mn)*
    - Např. pokud vzorek i text jsou složeny z opakování jednoho symbolu.
- Pro přirozené jazyky:
  - Empirické studie ukázaly, že pro délku vzorku `m>5` provádí algoritmus přibližně 0.24 až 0.3 porovnání z počtu znaků v prohledávaném textu. Jinými slovy, porovnává asi jednu čtvrtinu až jednu třetinu znaků prohledávaného textu.
  - Mnohem efektivnější než předchozí algoritmy.

BM algoritmus - varianta

- Při využití pouze prvního pravidla je potřeba rozlišit 2 případy při neshodě znaku:
  - Znak z textu se nachází ve vzorku vpravo od aktuální pozice (CharJump doporučuje posun vzorku zpět).
  - Znak z textu se nachází ve vzorku vlevo od aktuální pozice.
- Řešení:
  - Využití pole (`L`), které pro každý znak udává jeho nejpravější výskyt ve vzorku.
  - Pokud k neshodě dojde na indexu `posT` v textu `t`, pak novou pozici v textu, od které začne porovnání směrem doleva, lze určit takto:

```
l ← L[t[posT]]
posT ← posT + m - min(posP, 1+l)
```



Rabin-Karpův algoritmus

- Vyhledávání vzorku založené na hashování.
- Potřebujeme hashovací funkci, která m-ticím znaků (*m* je délka vzorku) přiřazuje čísla z množiny {0,…,N-1}.
- **Vyhledávání:**
  - posouváme okénko délky *m* po textu a počítáme hash pro danou část textu.
  - Je-li hash shodný s hashem vzorku, porovnáme danou část textu se vzorkem znak po znaku.
- Je-li hashovací funkce kvalitní, pak obvykle pro okénka, která neobsahují vzorek, bude hash jiný. Tím, že porovnáme pouze hashe, stačí nám pouze jedno porovnání pro každé okénko (neuvažujeme-li kolize).
- Problém: čas potřebný pro výpočet hashe
- Průměrný čas pro nalezení jednoho výskytu bude $\Theta(m+n)$

Rabin-Karpův algoritmus

- Potřebujeme hashovací funkci, kterou lze při posunu okénka o pozici doprava *rychle* (v konstantním čase) přepočítat.
- Lze použít polynom:

$$H(x_1, \dots, x_m) = (x_1 P^{m-1} + x_2 P^{m-2} + \cdots + x_{m-1} P^1 + x_m P^0) \; mod \; N$$

  - kde P je vhodná konstanta – nesoudělná s N a P^m musí být řádově větší než N, písmena považujeme za přirozená čísla.
- Při posunu okénka se hash změní takto:
- $$H(x_2, \dots, x_{m+1}) = (x_2 P^{m-1} + x_3 P^{m-2} + \cdots + x_m P^1 + x_{m+1} P^0) \; mod \; N$$
$$= (P \cdot H(x_1, \dots, x_m) - x_1 P^m + x_{m+1}) \; mod \; N$$

  - lze realizovat v konstantním čase (pokud si předpočítáme hodnoty P^m)

Rabin-Karpův algoritmus

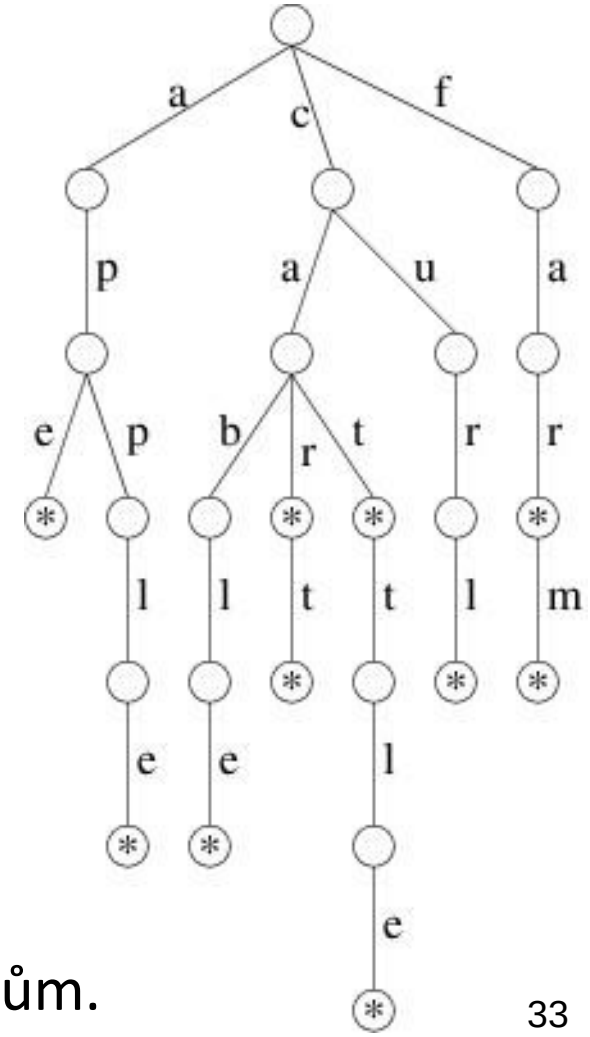
```
procedure RabinKarp (char *text, char *pattern)
// ohlásí všechny výskyty vzorku v textu
// P a M jsou vhodné konstanty hešovací funkce a máme
// předpočítáno P^m
  j ← H(pattern)                              // heš vzorku
  h ← H(text(0,patternLength-1))              // heš prvního okénka
  for i ← (0, textLength-patternLength):      // možné pozice okénka
    if j = h:                                 // shodné heše
      if SameCharacters(pattern, text (i, patternLength-1)):
        print i
    if i < textLength - patternLength:
                          // výpočet heše pro další pozici okénka
      h ← (P·h – t[i]·P^m + t[i+m]) mod N
```

Vyhledávání více vzorků

- Algoritmus Aho-Corasicková:
  - Rozšíření KMP algoritmu.
  - Využití písmenkového stromu.
- Využití konečných automatů.

Písmenkové stromy

- *Trie*, prefixové stromy.
- Struktura umožňující uložení slovníku (množiny slov – řetězců nad pevnou konečnou abecedou).
- Každému slovu lze přiřadit hodnotu.
- Zakořeněný strom, kde z každého vrcholu vedou hrany označené navzájem různými symboly abecedy.
- Vrcholům můžeme přiřadit řetězce tak, že přečteme všechny znaky na cestě z kořene do daného vrcholu.
- Označíme vrcholy odpovídající slovům a uložíme do nich hodnoty přiřazené klíčům.



Písmenkové stromy

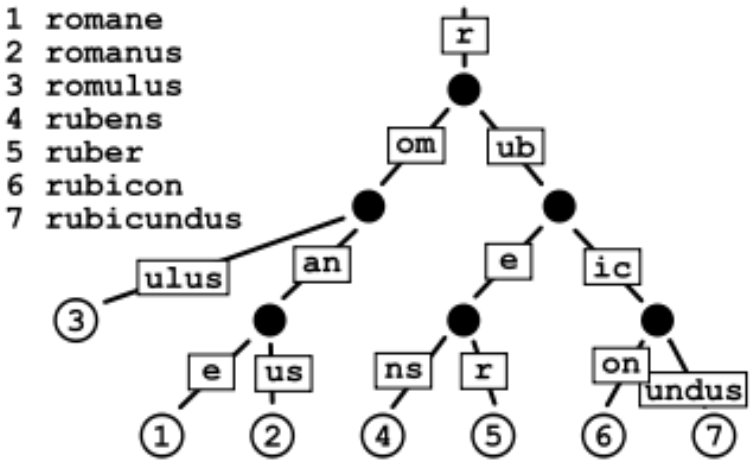
- Vrcholy v hloubce *h* odpovídají prefixům délky *h* uložených slov.
- Operace:
  - Vyhledávání – začneme v kořeni a následujeme hrany označené písmeny hledaného slova. Pokud existuje celá cesta a skončíme v označeném vrcholu, slovo je nalezeno. Kdykoliv hrana s daným písmenem chybí – neúspěšné hledání.
  - Vkládání – pokusíme se dané slovo vyhledat, kdykoliv chybí nějaká hrana, tak ji přidáme. Poslední vrchol označíme.
  - Mazání – rekurzivně tak, že nejprve procházíme stromem směrem dolů, na konci smažeme značku a cestou zpět mažeme vrcholy, které nemají žádné syny ani nejsou označené.
  - Složitost těchto operací je lineární vzhledem k počtu znaků daného slova

Písmenkové stromy

- Vnitřní reprezentace hran v trii:
  - Pomocí pole – v každém vrcholu trie bude pole o rozsahu $|\Sigma|$ položek.
  - Pomocí BVS nebo hashovací tabulky všech znaků, kterými může pokračovat aktuální prefix.
  - Transformace znaků abecedy do více znaků menší abecedy (např. abecedy se symboly 0, 1).
- Využití trie:
  - Uložení slovníku
  - Lexikografické řazení
  - Hledání nejdelšího společného prefixu
  - Inverzní vyhledávání v textu, …

Komprese trie

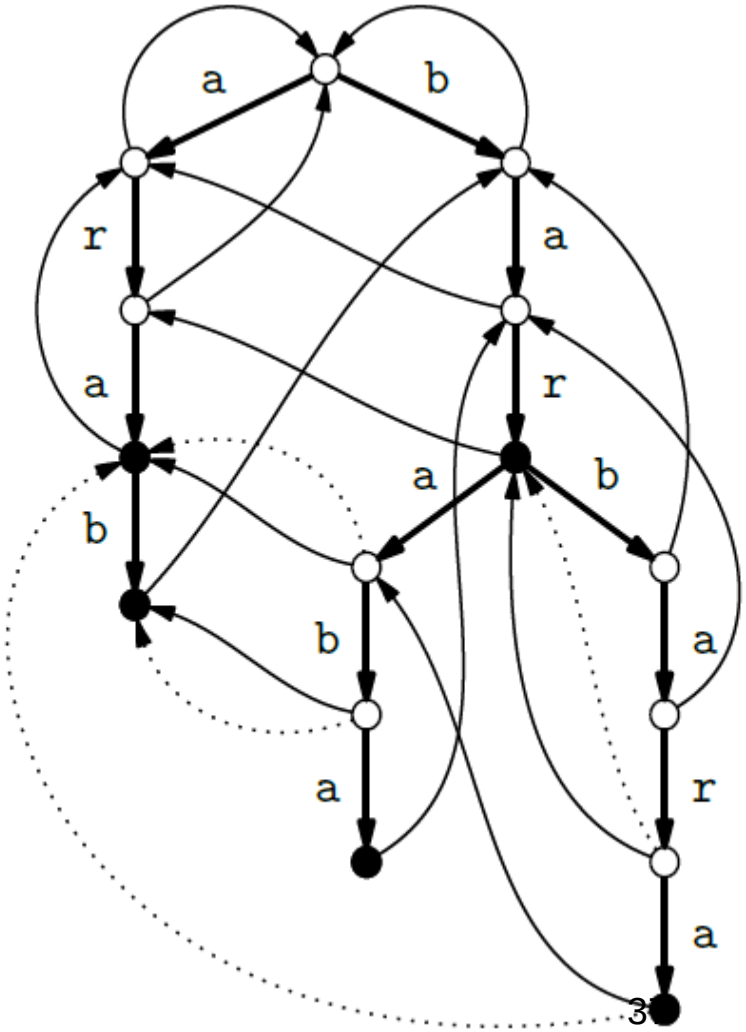
- Odstraňuje přebytečné vrcholy (ty, v nichž se slova nevětví)
- Hrana bude namísto písmene popsána celým řetězcem
- Komprimovanou trii lze převést zpět na normální trii



- Operace vkládání v komprimované trii se mírně komplikuje

Algoritmus Aho-Corasicková

- Rozšíření předchozího algoritmu pro vyhledávání více vzorků a hlášení všech výskytů
- Využívá vyhledávací automat – stavy odpovídají prefixům hledaných slov (písmenkový strom)
  - Koncové vrcholy – vrcholy, kde končí hledaná slova
  - Dopředné hrany – rozšíření prefixu o 1 znak
  - Zpětné hrany – stejné jako u KMP, ale mohou vést do jiných větví stromu
  - Zkratky – umožní ohlásit výskyt slova, který je sufixem jiného slova (na obr. tečkované hrany)



Algoritmus Aho-Corasicková

- Hledání slov:
  - Postupujeme automatem po dopředných hranách, pokud můžeme.
  - Nelze-li použít žádnou dopřednou hranu, vracíme se po zpětných hranách.
  - Pokud se dostaneme zpět až do kořene a ani zde nelze jít s daným symbolem žádnou dopřednou hranou, symbol je zahozen (je přečten nový symbol).
  - V každém stavu zkontrolujeme, zda neodpovídá konci slova. Pokud ano, ohlásíme výskyt. Z každého stavu pomocí zkratek nalezneme také všechny sufixy, které jsou také slovem a ohlásíme.

Algoritmus Aho-Corasicková

- Reprezentace automatu – pro každý stav potřebujeme tyto informace (stavy očíslujeme):
  - *Back(s)* – do kterého stavu vede zpětná hrana ze stavu s
  - *Shortcut(s)* – do kterého stavu vede zkratková hrana
  - *Word(s)* – zda v tomto stavu končí nějaké slovo (a jaké)
  - *Forward(s,x)* – kam vede dopředná hrana označená písmenem *x*
  - *Pozn.: Pro všechny hrany platí to, že pokud daná hrana neexistuje, reprezentujeme to hodnotou 0.*

Aho-Corasicková – jeden krok

```
int function ACStep (int state, char x)
  while Forward(state,x) = 0 and state ≠ root:
    state ← Back(state)
  if Forward(state,x) ≠ 0:
    state ← Forward(state,x)
  return state
```

Aho-Corasicková – vyhledávání

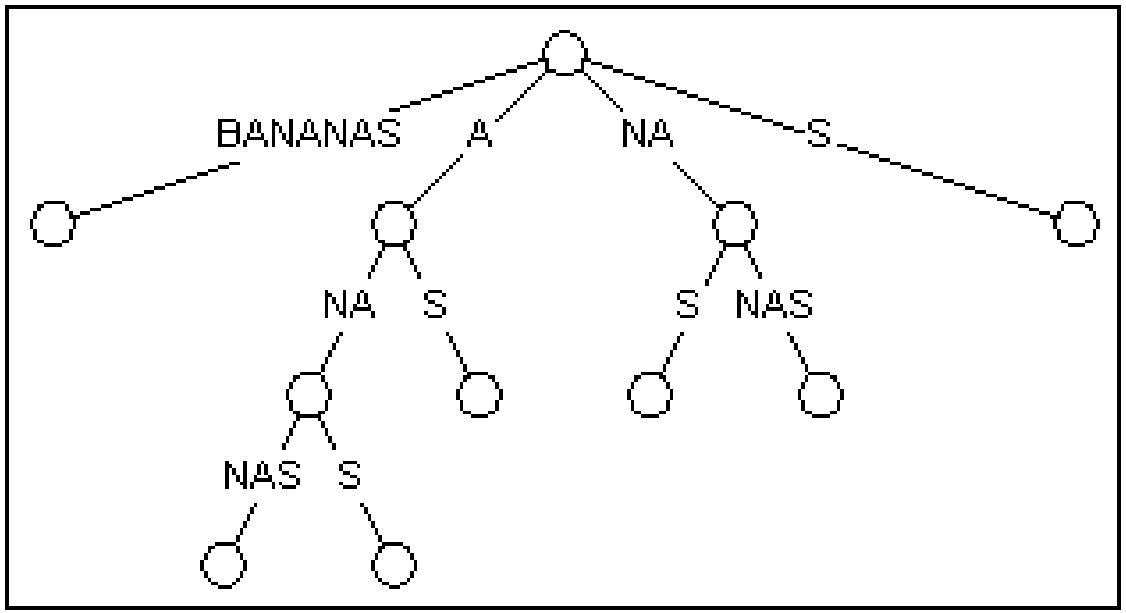
```
procedure ACSearch (char *t, int tl)
// používáme vytvořený automat, který považujeme za globální
  state ← root
  posT ← 0
  while (posT < tl):                  // pro každý symbol vstupu
    state ← ACStep(state,t[posT])     // proveď další krok
    j ← state
    while j ≠ 0:                      // dosažen konec slova?
      if Word(j) ≠ 0:
        print Word(j)                 // ohlášení výskytu
      j ← ShortCut(j)                 // zkontroluj sufixy
    posT ← posT + 1
```

Algoritmus Aho-Corasicková

- Konstrukce automatu se provádí po hladinách, protože zpětné hrany mohou vést křížem mezi jednotlivými větvemi stromu.
  - Princip zpětných hran je stejný jako u KMP
  - Kdykoliv je vytvořena zpětná hrana, je vytvořena také zkratka (vede-li zpětná hrana do stavu, kde žádné slovo nekončí, povede zkratka tam, kam vede zpětná hrana z tohoto stavu)
- Časová složitost: všechny vzorky jsou nalezeny v čase: ***O(n+m+v)***, kde *m* je zde součet délek všech hledaných slov a *v* je počet výskytů

Sufixový strom

- Komprimovaná trie všech sufixů daného slova (textu)
- Počet listů odpovídá délce slova
- Každý uzel má alespoň 2 syny
- Hrany jsou označeny neprázdnými řetězci (podřetězce slova)



Sufixový strom

- Využití:
  - Inverzní vyhledávání – z textu, který prohledáváme, vytvoříme sufixový strom. Pak můžeme vyhledávat libovolné slovo procházením stromu. Bude-li se slovo v textu nacházet, bude představovat prefix nějakého sufixu.
  - Nejdelší opakující se podslovo.
  - Nejdelší společné podslovo dvou slov.
  - Nejdelší palindromické podslovo.
- Konstrukce:
  - Lze sestrojit v lineárním čase a tedy i uvedené problémy lze řešit v lineárním čase.
  - Do prázdného stromu jsou postupně přidávány všechny prefixy daného slova (nový prefix vždy přidá symbol ke stávajícím sufixům a přidá tento sufix jako nový symbol).
  - Využití triků, které zajistí konstrukci v lineárním čase.