INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

7. LEKCE - VYHLEDÁVÁNÍ

prof. Ing. Radim Farana, CSc.

Ostrava 2013

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

7 VYHLEDÁVÁNÍ 3
7.1 Hešování (Hashing) 4
7.1.1 Lineární zkoušení (linear probing) 5
7.1.2 Dvojité hešování (double hashing) 6
7.2 Stromy s více potomky 8
7.2.1 2-3-4 stromy 9
7.3 Vyhledávání textových řetězců 10
7.3.1 Brute-Force algoritmus 10
7.3.2 Boyer-Moore algoritmus 11
POUŽITÁ LITERATURA 13

7 VYHLEDÁVÁNÍ

OBSAH KAPITOLY:

Hešování (Hashing)

Stromy s více potomky

Vyhledávání textových řetězců

MOTIVACE:

Vyhledávání ve velkých objemech strukturovaných i nestrukturovaných dat je velmi častá činnost, proto je dobré znát alespoň základní a přitom efektivní algoritmy vyhledávání.

CÍL:

Seznámit se s několika efektivními algoritmy vyhledávání ve strukturovaných i nestrukturovaných datech. Umět používat algoritmus vyhledávání hešováním, jak v podobě lineárního zkoušení, tak dvojitého hešování. Umět pro vyhledávání používat stromy s více potomky, jako jsou 2-3-4 stromy. Umět používat základní algoritmy pro vyhledávání v textech, jako je Brute-Force algoritmus nebo Boyer-Moore algoritmus

7.1 HEŠOVÁNÍ (HASHING)

Hlavní myšlenka:

U metody ***hešování*** je cílem najít umístění záznamu v tabulce tak, že provedeme aritmetickou transformaci klíče tak, abychom získali adresu v tabulce. Pravidla určující transformaci se nazývají ***hešovací funkce***.

Předpokládejme, že máme prvky s klíči obsahujícími celočíselné unikátní hodnoty v rozmezí **1 ..** ***N***. Pak můžeme použít tabulku s indexy v rozsahu **1 ..** ***N*** a hešovací funkci ***h*(*k*) = *k***. V tomto extrémním případě vyvstává problém pro velká ***N***. Je totiž nutné vytvořit pole o ***N*** prvcích, přičemž mnoho prvků pole může být neobsazeno. Tedy v tomto případě je možno hledaný prvek velice rychle nalézt v poli, avšak za velmi vysoké provize v paměti počítače. Druhým extrémem může být např. následující hešovací funkce :

$$h(k) = d_n + d_{n-1} + d_{n-2} + \cdots + d_1 + d_0 \qquad (7.1)$$

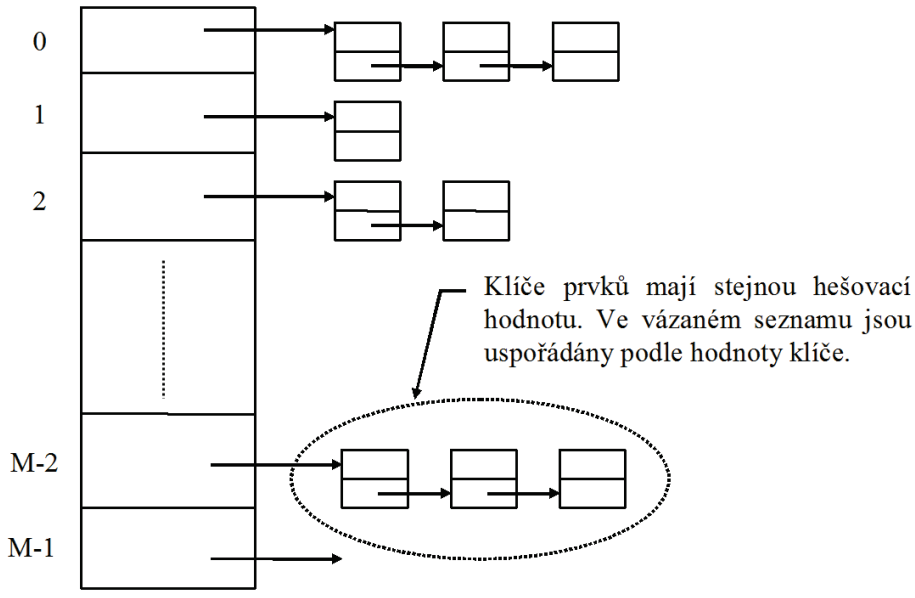
kde $d_n, d_{n-1}, \ldots, d_1, d_0$ jsou jednotlivé číslovky klíče.

Pak např.

h(1234) = h(4321) = h(2134) = h(1000432) = h(91)

V tomto druhém extrému se vytváří mnoho tzv. ***kolizí*** v hešovací tabulce. Jinými slovy mnoho různých klíčů má stejnou mapovací adresu do hešovací tabulky, danou hešovací funkcí. Dobrá hešovací funkce leží někde mezi uvedenými extrémy. Prakticky se např. používá hešovací funkce ***h(k) = k* mod *M***, kde ***k*** je celé kladné číslo a ***M*** je prvočíslo.

Samozřejmě, vždy může nastat kolize pro dva nebo více klíčů a tak je nutné použít tzv. ***kolizní metodu***. Jednou z metod může být použití např. dynamicky vázaného setříděného seznamu pro uchování prvků se stejnou mapovací adresou.



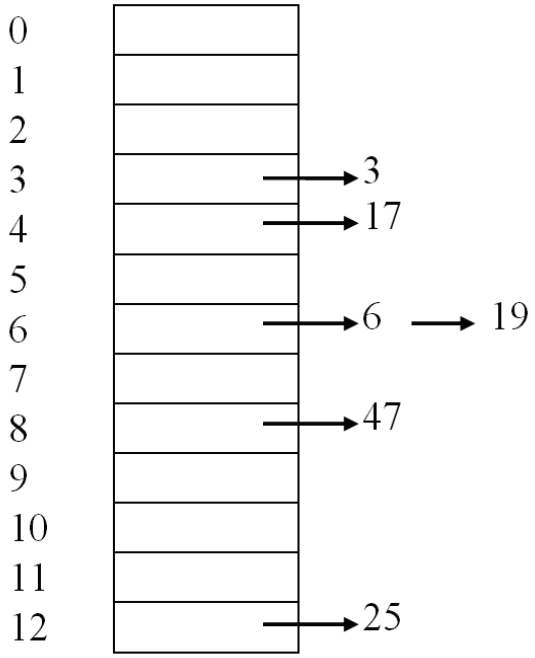
Obrázek 7.1 Použití dynamického seznamu pro řešení kolizí v hešovací tabulce

Hešovací tabulka pak má ***M*** ukazatelů na typ prvek.

Uvažujeme následující posloupnost klíčů: 3, 17, 25, 19, 47, 6 a *M* = 13

Rozbor algoritmu.

Pro náhodně vygenerované klíče je délka vázaného seznamu v průměru ***N/M***. Jelikož každý seznam je setříděn, pak musíme v průměru prohlédnout polovinu prvků pro nalezení konkrétního prvku. Tedy pro nalezení prvku potřebujeme v průměru ***N/2M*** iterací.



Obrázek 7.2 Příklad tabulky s dynamickým seznamem

Dalším způsobem, jak vyřešit kolize mezi adresami, je použití tzv. ***otevřeného adresování (open addressing)***. Tato metoda vychází z předpokladu, že při zaplňování hešovací tabulky jsou některé její prvky nezaplněny a jejich adresy lze právě použít pro vyřešení nastalých kolizí. Otevření adresování lze realizovat dvěmi algoritmy:

1. **lineární zkoušení (linear probing)**
2. **dvojité hešování (double hashing)**

7.1.1 Lineární zkoušení (linear probing)

Když vytváříme hešovací tabulku a dojde ke kolizi, vyhledáme nejbližší volnou pozici a vložení prvku provedeme na tuto pozici. Prohledávání začíná vždy od vypočtené adresy směrem k vyšším adresám.

Příklad: předpokládejme prvky s následujícími klíči: 1*, 19, 5*, 1**, 18, 3, 8, 9, 14, 7, 5**, 24, 1***, 13, 16, 12, 5***.

Tedy ***N = 17*** a dále zvolme ***M = 19*** (prvočíslo)

Pro hešování použijeme funkci ***h(k) = k mod M***

Tabulka 7.1 Plnění tabulky metodou lineárního zkoušení

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
19	1*	1**			5*													
19	1*	1**	3		5*	5**	7	8	9					14				18
19	1*	1**	3	1***	5*	5**	7	8	9	24				14				18
19	1*	1**	3	1***	5*	5**	7	8	9	24	5***	12	13	14		16		18

Tabulka ukazuje postup vkládání jednotlivých prvků do hešovací tabulky. Řádky postupně promítají obsah tabulky vždy do místa kolize. Další pokračování vkládání se pak děje, pro větší přehlednost, v dalším řádku.

Pokud nyní potřebujeme vyhledat prvek v tabulce, musíme nejdříve použít hešovací funkci k nalezení indexu (adresy) v tabulce a pokud hledaný prvek zde není, musíme použít metodu lineárního prohledávání směrem doprava.

7.1.2 Dvojité hešování (double hashing)

U metody dvojitého hešování je zvolen následující postup při kolizi dvou prvků v tabulce :

3. vypočti adresu v hešovací tabulce a v případě, že nedošlo ke kolizi, použij ji,
4. pokud došlo ke kolizi, použij druhou hešovací funkci pro výpočet inkrementu, který je přičten k adrese vypočtené v prvním kroku,
5. pokud dojde opět ke kolizi, opakuj krok 2.

Prakticky se jako první hešovací funkce používá nám již známá:

$$h1(k) = k \bmod M, \tag{7.2}$$

zatím co pro druhé hešování funkce:

$$h2(k) = M - 2 - (k \bmod (M - 2)) \tag{7.3}$$

Uvažujeme stejnou posloupnost prvků s klíči jako v minulém příkladu a dále uvažujeme následující hešovací funkci:

$$h1(k) = k \bmod 19$$

a

$$h2(k) = 17 - (k \bmod 17)$$

hodnoty hešovacích funkcí pro jednotlivé prvky jsou uspořádány do tabulky:

Tabulka 7.2 Tabulka hodnot hešovacích funkcí pro jednotlivé tříděné prvky

klíč	1*	19	5*	1**	18	3	8	9	14	7	5**	24	1***	13	16	12	5***
h1	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5
h2	16	15	12	16	16	14	9	8	3	10	12	10	16	4	1	5	12

Hešovací tabulka pak bude vypadat následovně:

Tabulka 7.3 Hešovací tabulka pro metodu dvojitého hešování

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
19	1*		3		5*	5***	7	8	9	5**	1***	12	13	14	24	16	1**	18

Poznámka:

Pro umístění prvku s klíčem 5*** je nutné pětinásobného použití hešovací funkce h2.

Rozbor algoritmu:

Algoritmus může mít na jedné straně velice dobrou výkonnost, avšak na straně druhé i velice nedostatečnou. To zvláště v případě, že nastává mnoho kolizí a metoda řešení kolizí klíčů není zrovna nejlepší. Taková situace nastane např. v případě, kdy je zvolena metoda ***kvadratických pokusů***, kdy hešovací funkce použitá pro řešení kolize má tvar $h_i = i^2 \bmod N$, kde i je číslo kolize. Při použití této metody může nastat situace, kdy kolize není vyřešena a volné místo v tabulce není nalezeno i přesto, že existuje.

Podívejme se nyní, jak odvodit průměrný počet pokusů o vložení nového klíče nebo jeho vyhledání v existující tabulce. Předpokládejme, že všechny klíče mají stejnou pravděpodobnost výskytu a hešovací funkce tyto klíče distribuuje rovnoměrně do buněk tabulky, jejíž velikost je ***N*** a obsahuje již ***k*** prvků. Pravděpodobnost, že zařazovaný prvek se podaří umístit na první pokus, je **1** – ***k/N***. Pravděpodobnost, že bude nutné použít pouze jeden sekundární pokus, je rovna:

$$P_2 = \frac{k}{N} \cdot \frac{n-k}{N-1} \tag{7.4}$$

Pravděpodobnost zásahu volného místa v tabulce na ***i***-tý pokus lze vyjádřit vztahem:

$$P_i = \frac{k}{N} \cdot \frac{k-1}{N-1} \cdot \frac{k-2}{N-2} \cdot \ldots \cdot \frac{k-i+2}{N-i+2} \cdot \frac{N-k}{N-i+1} \tag{7.5}$$

Počet pokusů k přidání **(*k* + 1)** klíče do tabulky o ***N*** prvcích je tedy :

$$E_{k-1} = \sum_{i=1}^{k+1} ip_i = 1\frac{N-k}{N} + 2\frac{k}{N}\frac{N-k}{N-1} + \ldots + (k+1)\left(\frac{k}{N} \cdot \frac{k-1}{N-1} \cdot \frac{k-2}{N-2} \cdot \ldots \cdot \frac{k-i+2}{N-i+2} \cdot \frac{1}{N-i+1}\right) \tag{7.6}$$

$$E_{k-1} = \frac{N-1}{N-k+1}$$

Budeme-li chtít vyčíslit střední hodnotu pokusů potřebných pro vyhledání klíče v tabulce, můžeme použít stejného vztahu jako pro výpočet středního počtu pokusů na přidání prvku do tabulky. Potom

$$E = \frac{1}{M}\sum_{k-1}^{M} E_k = \frac{N-1}{M}\sum_{k=1}^{M}\frac{1}{N-k+2} = \frac{N+1}{M}\left(H_{N+1} - H_{N-M+1}\right) \tag{7.7}$$

kde

$$H_N = 1 + \frac{1}{2} + \ldots + \frac{1!}{N} \tag{7.8}$$

Aproximací HN získáme vztah $H_N \cong \ln(N) + \gamma$,kde γ je ***Eulerova konstanta***. Dále pak substitucí

$$\alpha = \frac{M}{N+1} \tag{7.9}$$

dostaneme vztah:

$$E = \frac{1}{2}\left(\ln(N+1) - \ln(N-M+1)\right) = \frac{-1}{\alpha}\ln(1-\alpha) \tag{7.10}$$

Koeficient α vyjadřuje tzv. koeficient zaplnění tabulky a vyjadřuje poměr obsazených a volných míst v tabulce. Vyčíslíme-li pro několik hodnot α počet pokusů o vyhledání prvku a umístíme-li výsledky do tabulky, získáme:

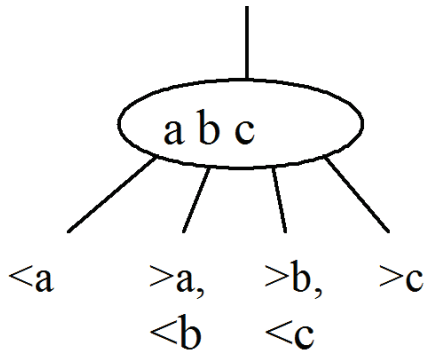
Tabulka 7.4 Závislost počtu pokusů na koeficientu zaplnění tabulky

α	E
0.10	1.05
0.25	1.15
0.50	1.39
0.75	1.85
0.90	2.56
0.95	3.15
0.99	4.66

Na základě získaných výpočtů lze prohlásit, že docela uspokojivých výsledků získáme i při 95% naplnění tabulky, kdy bude potřeba v průměru 3,15 pokusů o naplnění nebo získání hodnoty. Navíc je nutné si uvědomit, že uvedený počet pokusů není závislý na počtu prvků v tabulce, ale na koeficientu naplnění tabulky.

Výhodou principu hešování je relativně vysoká výkonnost algoritmu. Hešování však není možné vždy dobře implementovat. To zvláště v případech, kdy neznáme dopředu počet ukládaných prvků a tedy nemůžeme dobře odhadnout statický rozměr tabulky. Dále pak uvedený algoritmus není vhodný, když potřebujeme odebírat prvky z tabulky. Rušení prvků z tabulky je velmi složité, kromě metody řešící kolize zřetězením stejných klíčů pomocí lineárního dynamického seznamu.

7.2 STROMY S VÍCE POTOMKY



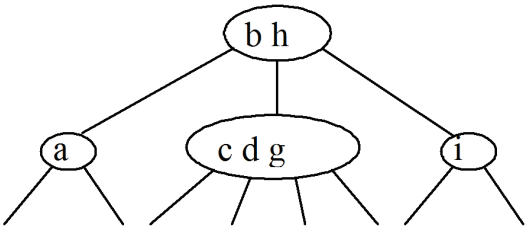
Obrázek 7.3 B-strom, m=4

Další velice často používanou metodou pro vyhledávání prvků je použití binárních vyhledávacích stromů. Binární stromy, jak bylo již uvedeno v kapitole věnované stromům, nejsou vhodné, protože může při jejich výstavbě dojít k vysoké nevyváženosti nebo dokonce k degradace v jednosměrně vázaný seznam. Jeden ze způsobů jak vyvážit binární strom je použít jeho modifikaci – ***AVL strom***. Třetím způsobem, jak vyvážit strom je umožnit, aby uzly mohly mít více než dva potomky. Místo binárních stromů se pak používají ***stromy s více potomky*** (***multi-way trees***). Ty pak obsahují uzly s ***n***-potomky a tedy ***n***-vazbami a uzel má pak ***n*** – **1** klíčů. Uzly s klíči menšími než ***a*** jsou vkládány do podstromu 1. Uzly s klíči mezi hodnotami ***a*** a ***b*** pak do podstromu 2. Podobně je tomu s prvky v rozmezí ***b*** a ***c***, které jsou vloženy do podstromu 3. Do podstromu 4 jsou pak vloženy prvky, které mají hodnotu klíče > než ***c***. Pro daný typ uzlů bude mít strom menší výšku než balancovaný binární strom, protože je schopen uchovat více klíčů v jednom uzlu. Stromy s více vazbami jsou tedy lehčeji vyvážitelné než binární stromy.

Jedním speciálním stromem je tzv. ***B-strom*** (***Bayerův strom***). B-strom stupně ***m*** má uzly, které mohou mít až ***m*** potomků. Nyní se podíváme na konkrétní případ, kdy ***m=4***, tedy B-stromy 4.stupně, které jsou často nazývány ***2-3-4 stromy***.

7.2.1 2-3-4 stromy

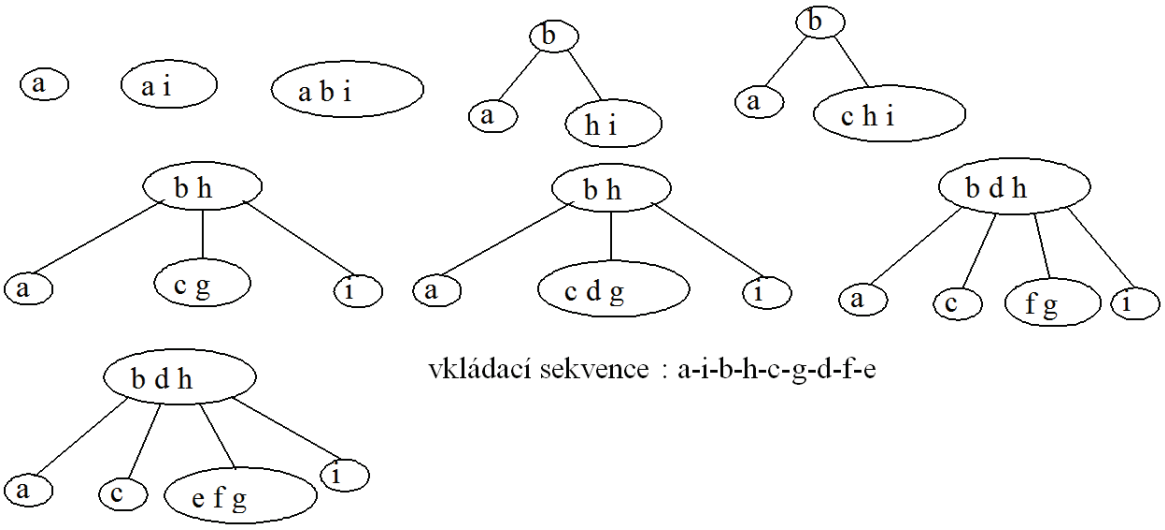
Název je odvozen od skutečnosti, že 2-3-4 strom může mít uzly s dvěmi, třemi nebo 4-mi potomky. Přidávání prvku do 2-3-4 stromu je odlišné od přidávání prvku do binárního stromu a



Obrázek 7.4 2-3-4 strom

vypadá následovně:

6. vyhledat místo, na které se má vložit nový uzel
7. pokud je místo vložení povrch (uzel) se dvěmi vazbami, stane se z něj uzel se 3-mi vazbami.
8. pokud je místo vložení uzel se 3-mi vazbami, stane se z něho uzel se 4-mi vazbami.
9. pokud je místo vložení uzel se 4-mi vazbami, je tento uzel už plný a je nutné jej rozdělit na dva uzly, každý se dvěma vazbami. Nový prvek je pak vložen do odpovídajícího uzlu, ze kterého se stane uzel se 3-mi vazbami.



Obrázek 7.5 Budování 2-3-4 stromu

Obrázek ukazuje, jak se rozděluje uzel se 4-mi vazbami. Prostřední klíč je vysunut směrem nahoru a stane se případně novým kořenem a další dva klíče jsou umístěny vpravo a vlevo od kořene a tvoří uzly se dvěmi vazbami. Rozložení uzlu se tedy děje vždy směrem nahoru a tím je zajištěno vyvážení stromu po celou dobu jeho výstavby.

Vkládání a tedy budování se sestává obecně ze dvou činností -

10. prohledání stromu seshora dolů
11. rozložení uzlů zespoda nahoru

Jak bylo vidět na obrázku, strom zůstává vyvážen i při použití známé patologické sekvence[1] klíčů.

7.3 VYHLEDÁVÁNÍ TEXTOVÝCH ŘETĚZCŮ

Jednou z častých úloh při zpracování informace je vyhledání tzv. vzoru v poli dat. Nejčastěji je uvedená úloha realizována jako vyhledání podřetězce (vzoru) v řetězci znaků. V uvedeném případě je pole dat jednorozměrné. Existují však úlohy, kdy pole dat je i vícerozměrné. Jako příklad můžeme uvést úlohu zpracování dvojrozměrného nebo dokonce trojrozměrného obrazu, kdy se snažíme například o nalezení jistého detailu na fotografii, nebo informaci z průmyslové kamery. Následovně uvedené algoritmy převážně řeší první typ úlohy, tedy úlohu vyhledání vzory v textovém řetězci.

7.3.1 Brute-Force algoritmus

Základní myšlenka tohoto algoritmu je velice jednoduchá a spočívá v prohledání každé případné pozice v daném textu na výskyt vzoru. Předpokládejme tedy, že textový řetězec, který se má prohledávat je uložen v poli znaků ***a*[1 .. *N*]** a hledaný vzor v poli ***p*[1 .. *M*]**. V následujícím funkci jsou použity dvě indexové proměnné ***i*** pro textový řetězec a ***j*** pro vzor. Funkce má dva formální parametry obsahující vzor a řetězec a dále pak vrací celočíselnou hodnotu určující nalezenou pozici vzoru v řetězci.

```
BruteForce(p : array of char, a : array of char): integer
  j=1
  i=1
  repeat
        if (a[i]=p[j])
              i=i+1
              j=j+1
        else
              i=i-j+2
              j=1
        end if
  until (j>M) OR (i>N)
  if (j>M)
        BruteForce = i-M
  else
        BruteForce = 0
  end if
end BruteForce
```

Rozbor algoritmu:

[1] Patologická sekvence způsobí vytvoření degenerovaného binárního stromu. Degenerovaný strom je takový, který je absolutně nevyvážený.

Brute-Force algoritmus je nejjednodušší ze série uvedených algoritmů, avšak také nejpomalejší. Algoritmus je citlivý na uspořádání vstupních dat. Jeho složitost je pro nejhorší případ (vzor není obsažen) velice jednoduše vyčíslitelná a je rovna ***O(M*N)***, zatím co pro nejlepší případ (vzor je hned na počátku řetězce) je rovna ***O(M+N)***.

7.3.2 Boyer-Moore algoritmus

Základní myšlenka algoritmu spočívá ve vybudování tabulky indexů posunutí. Její velikost je dána množinou znaků, které se mohou vyskytnout v prohledávaném a hledaném řetězci. Všechny prvky tabulky jsou implicitně inicializovány na hodnoty ***N***, která udává délku vzoru. Prvky pole odpovídající znakům obsažených ve vzoru jsou následovně nahrazeny hodnotami ***N – j***, kde ***j*** je pozice znaku v poli vzoru.

Vyhledávání vzoru v řetězci se provádí v inverzním pořadí postupně od znaku s indexem ***N*** tak dlouho, dokud není nalezen rozdílný znak. Pokud je takový rozdílný znak nalezen, pak dojde k použití tabulky indexů k výpočtu posunutí vzoru vzhledem k prohledávanému řetězci. Nedochází tedy k posunutí o jeden znak, jak tomu bylo u Brute-Force algoritmu. Posunutí u Boyer–Moore algoritmu je dáno znakem na pozici ***N*** v prohledávaném řetězci. Pokud tento znak ve vzoru neexistuje, dojde k posunutí vzoru vzhledem k aktuální pozici o celou jeho délku – ***N***, protože je zcela jasné, že jakékoliv menší posutí by skončilo následovaným neúspěšným vyhledáním. V případě, že tento znak ve vzoru existuje, dojde k posunutí vzoru tak, aby se stejné znaky v prohledávaném řetězci a vzoru kryly.

Následující program pro zjednodušení předpokládá, že používáme pouze velká písmena abecedy, tedy A-Z. Tabulka indexů posunutí je uložena v poli ***skip***, prohledávaný text v poli znaků ***a*** a vzor v poli ***p***.

```
BoyerMoore(p : array of char, a : array of char): integer
  REM inicializuj tabulku
  for i=0 to 31 do
        skip[i]=N
  end for
  REM napln prvky odpovidajici znakum ve vzoru
  for i=1 to N-1 do
        skip[p[j]-'A']=M-i;
  end for
  i=N+1
  REM zacatek vyhledavani
  repeat
        j=N+1
        k=i
        repeat
              j=j-1
              if (j=0)
                    BoyerMoore = k    REM vzor nalezen
                    exit function
              end if
              k=k-1
        until (p[j]<>a[k])
        i=i+skip[a[i-1]]              REM posun vzor
  until (i>M+1)
  BoyerMoore = 0
end BoyerMoore
```

Příklad:

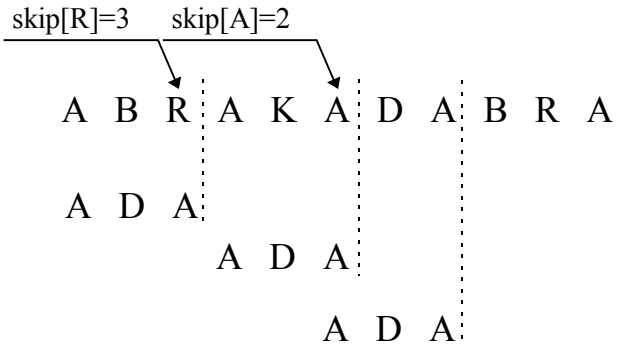
Vyhledejte slovo ADA ve slově ABRAKADABRA. (N = 3, M = 11.)

Tabulka skip bude obsahovat následující hodnoty:

Tabulka 7.5 Hodnoty

pozice (odpovídající znak abecedy)	hodnota
0 (A)	2
1 (B)	3
2 (C)	3
3 (D)	1
4 (E)	3
:	:
:	:
31 (Z)	3

Posuny slova ADA vzhledem k prohledávanému řetězci jsou pak:



Obrázek 7.6 Úpravy při odstranění uzlu – rotace

Rozbor algoritmu:

Boyer-Moore algoritmus je v současné době nejrychlejším algoritmem pro vyhledávání vzorů v řetězcích znaků. Proto je tak většinou implementován jako vyhledávací algoritmus pro textové řetězce v knihovnách různých programovacích jazyků. Jeho složitost je $O(M + N)$.

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ľ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ľ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5* – Uživatelská příručka. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms*. 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.