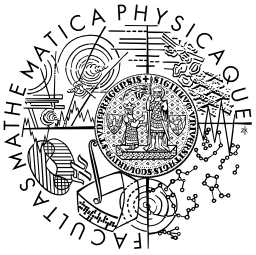

MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Václav Končický

Kompaktní popis adresářových stromů

202. Katedra aplikované matematiky

Vedoucí bakalářské práce:	Mgr. Martin Mareš, Ph.D.
Studijní program:	Informatika
Studijní obor:	Obecná informatika

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Poděkování.

Název práce: Kompaktní popis adresářových stromů

Autor: Václav Končický

Katedra: 202. Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, Ph.D., 202. Katedra aplikované matematiky

Abstrakt: Abstrakt. [doporučeno cca 80–200 slov; nejedná se o zadání práce]

Klíčová slova: klíčová slova [obvykle 3 až 5 klíčových slov či frází]

Title: Compact description of directory trees

Author: Václav Končický

Department: 202. Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D., 202. Department of Applied Mathematics

Abstract: Abstract.

Keywords: key words

Obsah

Úvod

1. Nesetřízené útržky

1.1 Adresářová struktura a stromy

V této práci se zaměříme primárně na adresářové stromy POSIXového světa. V tomto světě je každá položka adresáře odkazem (hard link) na *inode*, který již jednoznačně určuje vše kromě názvu. Kromě toho každý adresář obsahuje speciální odkazy `.` a `..`, které odkazují na sebe samotného a nadřazený adresář. Kromě těchto speciálních odkazů každý adresář může mít právě jeden odkaz.

Právě díky tomu se můžeme na libovolnou rozumnou adresářovou strukturu dívat jako na konečný zakořeněný strom – každý adresář je položkou právě jednoho jiného adresáře. V případě, že na soubor existuje více odkazů, budeme se ke každému z nich chovat jako k jinému souboru. Z toho již vyplývá, že každý odkaz má jednoznačně určenou cestu z počátku.

1.1.1 Terminologie

Pro definice pojmů, které budeme v práci používat, si vypůjčíme termíny z teorie grafů. Speciálně, na libovolný adresářový strom se budeme dívat jako na orientovaný zakořeněný strom, kde vrcholy jsou všechny odkazy, které jsou dosažitelné a orientované hrany ukazují vztah „adresář obsahuje odkaz“.

Abychom nepracovali s adresářovým stromem jako s abstraktním grafem, zavedeme si specializovanou terminologii.

Definice 1. *Nechť a je vrchol. Potom* položky $E(a)$ *jsou všichni bezprostřední synové vrcholu a. Podstrom zakořeněný v a budeme značit $T(a)$, striktní podstrom $T'(a) = T(a) \setminus a$.*

Pro každou položku $e \in E(a)$ je a nadřazeným adresářem $p(e)$.

Každý vrchol je určen svým názvem a odkazem na soubor. Proto, pokud mluvíme v kontextu adresářových stromů, budeme vrcholy rovněž nazývat *soubory* Z této definice se nám z jednoho souboru (daného svým inodem), na který odkazuje více vrcholů, stane více rozlišitelných souborů.

1.2 Požadavky na formát popisu

Při sestavování formátu popisu, ve které se reprezentuje adresářový strom, byly kladeny tyto požadavky:

- Jednoznačnost výsledných dat
- Minimalizace potřebného prostoru pro uložení celého stromu a času na jeho přečtení, minimalizace počtu průchodů a zpětných skoků

- Zpětná kompatibilita s dřívějšími verzemi programu a dopředná syntaktická kompatibilita

Protože každá adresářová struktura odpovídá stromu, můžeme k jeho popisu využít metody kódování stromů do seznamu. Pak není třeba si pro každý vrchol pamatovat celou jeho cestu.

Aby bylo toto stromové uspořádání jednoznačné, potřebujeme pro každý adresář a určit lineární uspořádání $T(a)$. Pro účely čtení nejlépe funguje preorder uspořádání, kde v každém adresářovém podstromu $T(a)$ je na prvním místě a a navíc $E(a)$ jsou uspořádané lexikograficky, pro jejich podstromy postoupíme rekurzivně.

Rekonstrukce cesty k danému souboru je potom v tomto uspořádání při lineárním průchodu velmi jednoduchá. Při průchodu pro každý podstrom narazíme na jeho kořen jako první, tudíž jde udržovat bez větších problémů aktuální větev celého stromu.

Toto uspořádání nám rovněž poskytuje velmi jednoduchý způsob, jak přeskakovat celý podstrom. Stačí si u jeho kořene zapamatovat adresu na konec podstromu nebo velikost jeho reprezentace.

Tyto volby nám již stačí k tomu, aby celý formát záznamu byl přečitelný na jeden lineární průchod.

Z těchto požadavků dále vyplývá, že za účelem šetření prostoru budeme preferovat binární formát oproti formátu textovému. Binární formát však nelze otevřít v libovolném textovém editoru a pak jej číst/upravovat. V tomto případě jsou úpravy naštěstí nežádoucí. Na druhou stranu to znamená, že pro jejich čtení bude potřeba použít program k tomu přímo uzpůsobený.

Nemá smysl se snažit dodržovat fixní šířku reprezentace všech dat. Pro mnoho dat nemáme zaručenou jejich velikost. Proto pro jejich reprezentaci v tomto případě musíme určit velikost jejich přihrádky a zde nastávají dva problémy.

Buď by byla zvolená velikost příliš malá, potom ale nemáme dost místa na uložení dat, nebo při nastavení příliš velkého rozsahu se použije zbytečně mnoho prostoru. Jako nejlepší řešení je proto použití kódování variabilní délky pro ta data, u kterých neznáme jejich rozsah velikostí předem, přestože si tím zkomplikujeme zápis i čtení.

1.3 Formát popisu

Syntaktická část je obecná pro všechna použití a snaží se být kompatibilní dopředně i zpětně. Naopak sémantická specifikace se snaží být kompatibilní pouze zpětně a závisí na verzi programu. Nejprve se podíváme na syntaxi.

Jak jsme již naznačili předem, celý formát záznamu bude binární. Ten se bude skládat ze čtyřech částí:

1. Hlavička popisu, obsahující:

- Identifikátor formátu souboru
- Metadata vztahující se na celou reprezentaci struktury
- Specifikace sloupečků pro každý záznam

2. Samotné záznamy vrcholů, obsahující povinné a podmnožinu nepovinných sloupečků

Pro zjednodušení zápisu proměnných si nadefinujeme speciální notaci.

Definice 2. *k:n značí binární řetězec k délky n. Pokud je navíc k složený z podřetězců $k_1, \ldots, k_m$, potom k lze rovněž zapsat jako $k\langle k_1, \ldots, k_m\rangle$:$n$. Podřetězce $k_1, \ldots, k_n$ mohou být uvnitř zapsány stejným způsobem.*

Pokud je binární řetězec k délky $8n, n > 0$ (tedy zarovnaný na celé bajty), lze rovněž použít značení k::n ekvivalentní s k:$8n$.

Příklad. Ukažme si nyní značení na několika jednoduchých příkladech:

- Binární zápis 32-bitového čísla i lze značit i::4 nebo i:32.
- 10-bitový řetězec r rozdělený na dva 5-bitové podřetězce r_1 a r_2 má značení $r\langle r_1{:}5, r_2{:}5\rangle{:}10$.
- Řetězec r o délce n bajtů lze zapsat jako r::n.

V tomto formátu souboru budeme pro nezáporné číslo k často používat speciální reprezentaci, která používá variabilní počet bajtů závisející na jeho velikosti, nejvýše však 64 bitů. Tomuto zápisu budeme říkat VarInt.

Pokud se binární zápis x vejde do $7n$ bitů, ale ne do $7(n-1)$ bitů, na jeho zápis použijeme n bajtů. Prvních n bitů prvního bajtu je $1^{n-1}0$, v případě $n = 9$ uvažujeme pouze 1^8. Zbytek nevyužitých bitů i v prvním bajtu je použit na zápis čísla x.

Počet bajtů, kolik zabere VarInt zápis čísla x, budeme značit $\|x\|$.

Jak vypadají všechny možné zápisy, lze vidět v tabulce 1.1.

Rozsah x	Bitů nejvýše	Počet bajtů $\|x\|$	Výsledný VarInt
$0 \ldots 2^7 - 1$	7	1	`0xxxxxxx`
$2^7 \ldots 2^{14} - 1$	14	2	`10xxxxxx` + 1 bajt
$2^{14} \ldots 2^{21} - 1$	21	3	`110xxxxx` + 2 bajty
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$2^{49} \ldots 2^{56} - 1$	56	8	`11111110` + 7 bajtů
$2^{56} \ldots 2^{63} - 1$	63	9	`11111111` + 8 bajtů

Tabulka 1.1: Všechny možné rozsahy VarIntu

1.3.1 Hlavička popisu

[Magic, metadata, specifikace povinných sloupečků.]

Celý soubor obsahuje na svém začátku svůj identifikátor – krátký řetězec identifikující třídu formátů a číslo identifikující jeho verzi.

Po tomto identifikátoru následuje seznam metadat. Každá položka metadat začíná svým sémantickým identifikátorem $h = \langle str{:}1{,}id\rangle{::}4$. Konec seznamu metadat nastává, když $h = 0^{32}$. V opačném případě následuje délka položky metadat k uložena ve VarIntu. Nakonec následuje k bajtů obsahujících obsah dané položky.

Interpretace metadat závisí pouze na sémantické hodnotě *id*. Pokud je pro aktuální verzi programu hodnota *id* neznámá, smí program tuto položku ignorovat. Je doporučeno v takovém případě vypsat chybovou hlášku včetně obsahu metadat. Bit *str* je naopak nápověda pro program, zda daná metadata lze interpretovat jako textový řetězec.

Jestliže se vyskytne více instancí metadat se stejným *id* (nezáleží na bitu *str*), budou platná pouze ta poslední.

Nyní následují seznamy povinných a nepovinných sloupečků každého záznamu. Obojí je specifikováno stejným způsobem:

Každá položka v seznamu sloupečků začíná identifikátorem $h = \langle type{:}2, str{:}1, id\rangle{::}4$. Stejně, jako u specifikace metadat, $h = 0^{32}$ značí konec seznamu. *type* značí, o jaký typ sloupečku se jedná:

- 00 – data fixní šířky. V takovém případě následuje ještě VarInt k značící délku dat.
- 01 – číslo ve VarIntu.
- 10 – data variabilní délky.
- 11 – v současné verzi formátu nevuyžito.

Tyto tři hlavní typy jsou dostatečně obecné, aby šla zaznamenat libovolná data. Bližší specifika typu sloupečku jsou závislá na jeho sémantickém významu podle *id*. Význam bitu *str* je stejný, jako u specifikace metadat.

Pokud se v tomto případě vyskytne více sloupečků se stejným *id*, každá instance se vyskytuje v záznamech, pouze data z posledního výskytu jsou platná. Program navíc není povinen podporovat všechny dvojice (*id*,*type*). V případě neznámého *id* nebo dvojice (*id*, *type*) musí program stále číst sloupečky (jelikož se stále v záznamech vyskytují), ale jejich obsah smí zahodit.

Příklad. Nyní se podívejme na ukázkovou hlavičku. Mějme dvě položky metadat, první je řetězec `Hello World` identifikovaný číslem 42. Druhá položka je 64-bitové číslo `0xdeadbeef0badf00d` identifikované číslem 2018. Potom seznam metadat je uložen takto:

|0x80|0|0|42| |11| |Hello World| |0|0|2018| |8| |0xdeadbeef0badf00d| |0|0|0|0|

[Zlepšit formátování příkladu]

1.3.2 Formát záznamu

Po hlavičce následuje uspořádaný seznam jednotlivých záznamů souboru. Nechť C je seznam povinných a O seznam nepovinných sloupečků. Každý sloupeček v seznamech si pamatuje svůj *id*, *type* a k, pokud $type = 00$.

Záznam začíná svou délkou n ve VarIntu. Samotné $\|n\|$ se do této délky nezapočítává.

Nyní pro každý sloupeček z C následují jeho data v záznamu. Pro data fixní či variabilní délky jednoduše následuje k bajtů dat. U dat variabilní délky je navíc před těmito k daty uloženo k jako VarInt. Pro $type = 01$ je daty sloupečku číslo ve VarInt podobě.

Po seznamu dat povinných sloupečků následuje bitové pole q velikosti $\lceil |O|/8 \rceil$, kde i-tý nejnižší bit q značí, zda záznam obsahuje data i-tého nepovinného sloupečku v seznamu O.

Poté následují data nepovinných sloupečků O. Jestliže $q[i] = 0$, není uloženo nic. Jinak jsou uložena data stejným způsobem, jako u dat povinných sloupečků.

1.4 Hešování vrcholů

Abychom mohli rozpoznávat, že se podstromy změnily vůči sobě, potřebujeme způsob, jak je od sebe rozpoznat. Názvy jsou hlavní klíče, v případě rozdílných názvů považujeme tyto vrcholy za úplně nezávislé. Pro rozpoznání, zda se změnily stejné vrcholy, proto potřebjeme nějakou signaturu.

Ta se skládá ze dvou částí. První část jsou metadata, jakožto typ vrcholu, velikost podstromu, apod. Druhá, mnohem důležitější část, je signatura samotného obsahu souboru. Pro symbolické linky budeme jejich obsah uvažovat jejich cílovou adresu. O tom, jak budeme uvažovat obsah adresářů, povíme více v sekci 1.4.2.

Ne všechny vrcholy ale mají dobře definovaný obsah. Roury a zařízení, které se mohou v POSIXovém souborovém systému nacházet, mají velmi pomíjivý obsah. Navíc se může stát, že při čtení obsahu souboru nastane chyba a nebudeme schopni jej celý přečíst. Z tohoto důvodu se může u některých vrcholů obsah nevyskytovat.

Protože obsahy souborů mohou být potenciálně velmi velké, nemůžeme je používat přímo. Z tohoto důvodu budeme obsahy hešovat a poté porovnávat pouze tyto heše.

1.4.1 Výběr hešovací funkce

Heše obsahu, kromě metadat, budou jediný způsob, jak rozpoznat rozdílné obsahy mezi dvěma verzemi vrcholu. Tudíž je velmi důležité vybrat takovou hešovací funkci, ve které je pravděpodobnost, že nastane kolize, takřka nulová.

Zamiřme tudíž ke kryptografickým hešům, od kterých tuto vlastnost očekáváme. Podívejme se speciálně na tyto kandidáty:

- MD5 byla populární funkce na počítání signatury obsahu, avšak se již dávno ukázalo, že ji lze velmi snadno rozbít [Wang a Yu (2005)]. Tudíž je nevhodná.
- SHA-1 je rovněž velmi rozšířená funkce, kterou například používá verzovací systém git. Bohužel i na tuto funkci se nedávno podařilo úspěšně zaútočit [Stevens a kol. (2017)].
- SHA-2 je novější množina hešovacích funkcí, která vychází z SHA-1.
- SHA-3 je nejmladší z rodiny hešovacích funkcí SHA, jenž, na rozdíl od svých předků, používá permutaci Keccak. [Dworkin (2015)]

[Přidat výběr SHA-256, paralelní hešování, Sakura]

1.4.2 Hešování adresářů

2. Algoritmy pracující s reprezentací

Nyní se podíváme na algoritmy, které čtou nebo vytvářejí reprezentaci adresářového stromu.

Protože v mnoha případech budeme pracovat se soubory jako s proudem dat, zavedeme si jednoduché značení – pokud chceme načíst n bajtů z proudu f do proměnné x, výslednou operaci označíme $x \xleftarrow{n} f$. V případě, že chceme načíst VarInt, použijeme analogicky značení $x \xleftarrow{\text{VarInt}} f$. V případě zápisu do souboru f prohodíme směr šipky.

2.1 Čtení a zápis záznamů

Nejprve si ukážeme, jakým způsobem budeme číst a zapisovat jednotlivé sloupečky. Postup vyplývá přímočaře ze specifikace formátu souboru.

Procedura NačtiSloupeček
Vstup: Vstupní soubor f, sloupeček $c = (id, k)$

1. Pokud $k =$ VarInt: ◁ *Jedná se o číslo?*
2. $\quad d \xleftarrow{\text{VarInt}} f$
3. $\quad k = \|d\|$
4. Jinak:
5. $\quad$ Pokud $k = ?$: ◁ *Data variabilní délky?*
6. $\quad\quad k \xleftarrow{\text{VarInt}} f$
7. $\quad d \xleftarrow{k} f$ ◁ *Načteme data podle délky k*

Výstup: Data sloupečku $id \to (k, d)$

Zápis sloupečku je analogický k jeho čtení. Přečtení a zápis jednoho sloupečku přečte $\mathcal{O}(k)$ bajtů a kromě toho provede konstantní množství práce.

Přečtení jednoho celého záznamu znovu vyplývá přímo z formátu uložené reprezentace. Zde vidíme obecný algoritmus, který vrací množinu dat D, kde klíč je id a hodnota je dvojice (k, d), kde d jsou samotná data a k je jejich délka.

Procedura NačtiZáznam
Vstup: Vstupní soubor f, seznamy sloupečků C a nepovinných sloupečků O

1. $n \xleftarrow{\text{VarInt}} f$ ◁ *Načteme délku celého záznamu*
2. Pro každou položku $c = (id, k) \in C$:
3. $\quad$ Do D přidej NačtiSloupeček(f, c)
4. $\quad n \leftarrow n - k$
5. $i \leftarrow 0$
6. Pokud $O \neq \emptyset$:

7. $q \xleftarrow{\lceil |O|/8 \rceil} f$ ◁ *Načteme bitové pole nepovinných sloupečků*
8. Pro každou položku $c = (id, k) \in O$:
9. Pokud i-tý nejnižší bit q je 1:
10. Do D přidej NačtiSloupeček(f, c)
11. $n \leftarrow n - k$
12. $i \leftarrow i + 1$

Výstup: Množina dat D

Celkový čas, který zabereme touto operací, je $\mathcal{O}(n+I(|C|+|O|))$. Člen n dostáváme, protože čteme $\mathcal{O}(n)$ bajtů z f. Dále závisí na použití datové struktury na reprezentaci množiny dat, $I(x)$ je čas strávený na vložení dat do množiny a $F(x)$ je její vyhledání.

Zápis záznamu je podobný jeho čtení, avšak je složitější. Protože délku záznamu n ukládáme jako VarInt, závisí $\|n\|$ na délce zbytku záznamu. Proto je potřeba si n spočítat předem. Protože při výpočtu n zkoumáme nepovinné sloupečky, můžeme při tom i rovnou spočítat q.

Procedura SpočtiDélkuAMasku

Vstup: Seznam sloupečků C, seznam nepovinných sloupečků O, množina dat D

1. $n \leftarrow \lceil |O|/8 \rceil, q \leftarrow 0^{\lceil |O|/8 \rceil}$ ◁ *Připočteme délku bitmasky q*
2. Pro každé $c = (id, k_0) \in C$:
3. pro $id \to (k, \dots) \in D$: ◁ *Najdeme data sloupečku c*
4. $n \leftarrow n + k$
5. Pokud $k_0 =?$:
6. $n \leftarrow n + \|k\|$ ◁ *Pro data variabilní délky započteme i* $\|k\|$
7. $i \leftarrow 0$
8. Pro každé $c = (id, k_0) \in O$:
9. Pokud $id \to (k, \dots) \in D$: ◁ *Existují data nepovinného sloupečku?*
10. Nastav i-tý nejmenší bit q na 1
11. $n \leftarrow n + k$
12. Pokud $k_0 =?$: $n \leftarrow n + \|k\|$
13. $i \leftarrow i + 1$

Výstup: Délka záznamu n, bitové pole q

Spočítat n a q tedy umíme provést v čase $\mathcal{O}(n + F(|C| + |O|)$. F značí složitost vyhledání dat v D. Protože při zapisování nepotřebujeme vkládat do D sloupečky, jen je pro jejich zápis vyhledáváme, platí stejná složitost i pro zápis.

Obecně se tedy pro reprezentaci D dá využít hešovací tabulka. V takovém případě bude I i F průměrně konstantní složitost. Pokud navíc bude počet sloupečků nulové délky nejvýše konstantně mnoho, dostaneme se pro čtení a zápis jednoho záznamu na optimální složitost $\mathcal{O}(n)$.

Implementační detaily

Náš program je mnohem konkrétnější – pouze některé sloupečky mohou být nepovinné a předem očekáváme množinu všech možných sloupečků $\mathcal{C}$.

Celá množina dat bude jedna velká statická struktura, obsahující ukazatele na všechna možná data. Výjimkou budou čísla uložena přímo. Data variabilní délky budou kromě ukazatele obsahovat informaci o jejich délce.

Pro zaznamenaní, které nepovinné sloupečky daný záznam obsahuje, se použije bitové pole s jednotlivými bity přímo určenými daným datům.

Každý specifikovaný sloupeček (povinný i nepovinný) si kromě svého typu a id bude pamatovat offset na datovou a délkovou položku ve struktuře, přičemž pro čísla a data fixní délky bude délková položka prázdná. Navíc nepovinné sloupečky si budou pamatovat číslo bitu odpovídající bitové masce ve struktuře.

Při čtení a zápisu záznamu dostane procedura ukazatel na strukturu. Pro přístup či zápis se poté spočítá místo v paměti, určené pro čtení nebo zápis, pomocí ukazatele na strukturu plus offset.

Navíc v případě, když ke čtení záznamu dostaneme $c \notin \mathcal{C}$, v jeho specifikaci si budeme pamatovat jeho typ a informaci, že je neznámý. Poté se při jeho čtení přeskočí zápis do reprezentace.

S touto implementací je časová složitost F i I konstantní worst case. Naopak paměťová složitost je $\mathcal{O}(n+|\mathcal{C}|)$, i když v reprezentaci použijeme jen malou podmnožinu $\mathcal{C}$. Proto se tato reprezentace hodí pouze pro malé $|\mathcal{C}|$.

2.2 Čtení reprezentace

Naše reprezentace je velmi optimalizovaná na jednoduché čtení. V ní je celý adresářový strom zaznamenaný v preorder uspořádání. Navíc každá důležitá informace je vždy dostupná předem.

Díky tomuto stačí na datech akorát provést jeden lineární sken. Paměťové nároky jsou proto kromě pamatování potřebné části stromu (v mnoha případech stačí si pamatovat aktuální větev) velmi nízké – potřebujeme si ještě pamatovat akorát seznam povinných a nepovinných sloupečků, případně nějaká metadata navíc.

Nyní se pojďme podívat na typické přečtení a následné zpracování reprezentace. Označíme si N jako délku celého souboru.

Na začátku zpracujeme hlavičku, jejíž délka je H. Nejprve algoritmus ze vstupního souboru f přečte magické číslo. Tím se přesvědčí, že opravdu pracuje se správným formátem reprezentace adresářového stromu.

Poté jednoduchou smyčkou začne číst a zpracovávat metadata podle jejich konkrétní sémantiky:

1. $id \xleftarrow{4} f$
2. Pokud $id = 0$: konec
3. $n \xleftarrow{\textsc{VarInt}} f$
4. $m \xleftarrow{n} f$
5. Zpracuj metadata (id, m)
6. Vrať se na začátek

Po seznamu metadat ihned následuje specifikace povinných i nepovinných sloupečků C a O. Jejich čtení je totožné, lze proto pro jejich načtení použít stejný cyklus:

1. $I \xleftarrow{4} f$
2. Pokud $I = 0$: konec
3. $type \leftarrow I[30 \ldots 31]$ ◁ *Uložíme horní dva bity jako typ dat*
4. $id \leftarrow I[0 \ldots 28]$
5. Pokud $type = 00$: ◁ *Fixní délka*
6. $\quad n \xleftarrow{\textsc{VarInt}} f$
7. $\quad$ Do seznamu sloupečků přidej (id, n)
8. Pokud $type = 01$:
9. $\quad$ Do seznamu sloupečků přidej $(id, \textsc{VarInt})$ ◁ *Číslo ve* VarInt
10. Pokud $type = 10$:
11. $\quad$ Do seznamu sloupečků přidej $(id, ?)$ ◁ *Variabilní délka*
12. Vrať se na začátek

Celé přečtení hlavičky tedy umíme provést jedním lineárním skenem f a strávime nad tím optimální čas $\mathcal{O}(H)$. V tomto okamžiku máme zpracovaná všechna metadata i C, O. Můžeme tedy začít zpracovávat samotný adresářový strom.

Ukážeme si obecný postup, jak zpracovat libovolný podstrom prohledáváním do hloubky, jehož kořenem je aktuálně přečtený záznam. Budeme již předpokládat, že máme proceduru ZpracujZáznam, jenž načte jeden záznam a převede jej do reprezentace, která se nám v danou chvíli nejvíce hodí.

Algoritmus ZpracujPodstrom
Vstup: Aktuálně přečtený kořen podstromu x, další konkrétnější data
Výstup: Velikost podstromu V, další konkrétnější data

1. $V \leftarrow |T(x)|$
2. Proveď konkrétnější zpracování před rekurzí
3. Dokud $V > 1$:
4. $\quad y \leftarrow$ ZpracujZáznam
5. $\quad p(y) \leftarrow x$ ◁ *Pokud potřebujeme, zapamatujeme si otce*
6. $\quad V_y, \ldots \leftarrow$ ZpracujPodstrom$(y, \ldots)$
7. $\quad$ Proveď konkrétnější zpracování mezi rekurzí
8. $\quad V \leftarrow V - V_y$
9. Proveď konkrétnější zpracování po rekurzi
10. Vrať $|T(x)|, \ldots$

Zpracování celého stromu pak započneme zpracováním jeho kořene, tedy prvního záznamu, a zavoláním ZpracujPodstrom na něm. Složitost tohoto algoritmu

poté závisí na konkrétním zpracovávání. Stále však získáváme velmi hezkou vlastnost, že soubor s reprezentací čteme jen jednou jako proud dat.

Implementační detaily čtení hlavičky

Už při čtení sloupečků jsme nepracovali obecně, nýbrž jsme očekávali množinu známých sloupečků $\mathcal{C}$. Pro naše účely navíc máme všechny identifikátory sekvenční. Proto $\mathcal{C}$ reprezentujeme jako pole, jehož prvky jsou reprezentované stejným způsobem, avšak délka bude nedefinovaná.

Pokud některé *id* neznamenají nic, nebo se přestaly používat, v poli máme záznam, že toto *id* je neplatné.

Při čtení povinných či nepovinných sloupečků poté přistoupíme k danému indexu v poli. Pokud je platná, danou specifikaci zkopírujeme a nastavíme jí délku podle informace v souboru. Jinak vytvoříme generickou specifikaci se správným typem a délkou a označíme ji jako neznámou.

2.3 Sestavení reprezentace

Nyní již umíme libovolnou reprezentaci stromu přečíst a zpracovat, přičemž to celé umíme provést proudově. Na druhou stranu, sestavení této reprezentace takto jednoduché není.

Výslednou reprezentaci potřebujeme mít uspořádanou pre-order. Avšak v případě, že existuje sloupeček, jehož obsah je závislý na podstromu, potřebujeme nejprve zpracovat jeho podstrom. Z toho vyplývají dvě možnosti, jak postupovat:

1. Budeme si pamatovat celý podstrom. Jakmile je celý zpracovaný, projdeme jej v pre-order uspořádání a v něm jej uložíme.
2. Podstrom uložíme v pořadí zpracovaných vrcholů a následně jej externě přeuspořádáme do správného uspořádání.

První možnost bohužel v nepřipadá v úvahu. Disky mohou stále být řádově mnohem větší, než RAM, tudíž celý adresářový strom může být potenciálně tak rozsáhlý, že by se nám jeho reprezentace nemusela vejít do RAM.

Druhá možnost zase znamená, že se nám nepodaří celou reprezentaci uložit jedním průchodem. To navíc také implikuje, že si budeme navíc muset pamatovat pre-order pořadí každého vrcholu.

Skutečně, pro naše konkrétní požadavky máme alespoň dvě závislosti kořene na jeho podstromu – hešování adresářů a informace o velikosti podstromu.

Při sestavování proto rozdělíme celý algoritmus na dva hlavní kroky. První krok zpracuje všechny nezávislé sloupečky a vypočítá pre-order uspořádání. Druhý krok poté spočítá heše adresářů a případně další závislé hodnoty.

2.3.1 Hešování obyčejných souborů

Tento krok je jediný, který pracuje s reálným adresářovým stromem.

Závěr

Seznam použité literatury

Dworkin, M. J. (2015). Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A. a Markov, Y. (2017). The first collision for full sha-1. In *Annual International Cryptology Conference*, pages 570–596. Springer.

Wang, X. a Yu, H. (2005). How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer.

[Až budou citace, odstranit
nocite.]

Seznam obrázků

Seznam tabulek

[U matematických prací může být lepší přemístit seznam tabulek na začátek práce.]

Seznam použitých zkratek

[U matematických prací může být lepší přemístit seznam zkratek na začátek práce.]

A. Přílohy

[Přílohy k bakalářské práci, existují-li. Každá příloha musí být alespoň jednou odkazována z vlastního textu práce. Přílohy se číslují.]

[Do tištěné verze se spíše hodí přílohy, které lze číst a prohlížet (dodatečné tabulky a grafy, různé textové doplňky, ukázky výstupů z počítačových programů, apod.). Do elektronické verze se hodí přílohy, které budou spíše používány v elektronické podobě než čteny (zdrojové kódy programů, datové soubory, interaktivní grafy apod.). Elektronické přílohy se nahrávají do SISu a lze je také do práce vložit na CD/DVD. Povolené formáty souborů specifikuje opatření rektora č. 13/2017.]

A.1 První příloha