Gnarley Trees

Katka Kotrlová* Pavol Lukča† Viktor Tomkovič‡ Tatiana Tóthová§

Školiteľ: Jakub Kováč[1]¶

Katedra informatiky, FMFI UK, Mlynská Dolina, 842 48 Bratislava

Abstrakt: V tomto článku prezentujeme našu prácu na projekte Gnarley Trees, ktorý začal Jakub Kováč ako svoju bakalársku prácu. Gnarley Trees je projekt, ktorý má dve časti. Prvá časť sa zaoberá kompiláciou dátových štruktúr, ktoré majú stromovú štruktúru, ich popisom a popisom ich hlavných výhod a nevýhod oproti iným dátovým štruktúram. Druhá časť sa zaoberá ich vizualizáciou a vizualizáciou vybraných algoritmoch na týchto štruktúrach.

Kľúčové slová: Gnarley Trees, vizualizácia, algoritmy a dátové štruktúry

1 Úvod

Ako ľudia so záujmom o dátové štruktúry sme sa rozhodli pomôcť vybudovať dobrý softvér na vizualizáciu algoritmov a dátových štruktúr a obohatiť kompiláciu Jakuba Kováča (Kováč, 2007) o ďalšie dátové štruktúry. Vizualizujeme rôznorodé dátové štruktúry. Z binárnych vyvažovaných stromov to sú *finger tree* a *reversal tree*, z háld to sú *d-nárna halda*, *ľavicová halda*, *skew halda* a *párovacia halda*. Taktiež vizualizujeme aj *problém disjuktných množín (union-find problém)* a *písmenkový strom (trie)*.

Okrem vizualizácie prerábame softvér, doplnili sme ho o históriu krokov a operácií, jednoduchšie ovládanie a veľa iných vecí, zlepšujúcich celkový dojem. Softvér je celý v slovenčine a angličtine a je implementovaný v jazyku `Java`.

1.1 Vizualizácia

Dátové štruktúry a algoritmy tvoria základnú, prvotnú časť výučby informatiky. Vizualizácia algoritmov a dátových štruktúr je grafické znázornenie, ktoré abstrahuje spôsob ako algoritmus a dátové štruktúry pracujú od ich vnútornej reprezentácie a umiestnení v pamäti. Je teda vyhľadávaná a všeobecne rozšírená pomôcka pri výučbe. Výsledky výskumov ohľadne jej efektívnosti sa líšia, od stavu „nezaznamenali sme výrazné zlepšenie“ po „je viditeľné zlepšenie“. (Shaffer et al., 2010)

Rozmach vizualizačných algoritmov priniesla najmä Java a jej fungovanie bez viazanosti na konkrétny operačný systém. Kvalita vizualizácií sa líši a keďže ide o ľahko naprogramovateľné programy, je ich veľa a sú pomerne nekvalitné. V takomto množstve je ťažké nájsť kvalitné vizualizácie. Zbieraním a analyzovaním kvality sa venuje skupina AlgoViz, ktorá už veľa rokov funguje na portále `http://algoviz.org/`.

Zaujímavé je pozorovanie, že určovanie si vlastného tempa pri vizualizácií je veľká pomôcka. Naopak, ukazovanie pseudokódu alebo nemožnosť určenia si vlastného tempa (napríklad animácia bez možnosti pozastavenia), takmer žiadne zlepšenie neprináša. (Shaffer et al., 2010; Saraiya et al., 2004)

Motivácia

Z vyššie uvedeného je jasné, že našou snahou je vytvoriť kvalitnú kompiláciu a softvér, ktorý bude nezávislý od operačného systému, bude vyhovovať ako pomôcka pri výučbe ako aj pri samoštúdiu a bude voľne prístupný a náležite propagovaný. Toto sú hlavné body, ktoré nespĺňa žiaden slovenský a len veľmi málo svetových vizualizačných softvérov. Našou hlavnou snahou je teda ponúknuť plnohodnotné prostredie pri učení.

2 Rozšírenie predošlej práce

jednotlive vizualizacie a implementovane ficurie co sa zmenilo od bakalarky? zoomovanie, komentare, tree layouty, historia a nove ds

3 Vyvážené stromy

uz boli, pribudli

*`katkinemail,ktorýchcezverejniť`
†`palyhomail`
‡`viktor.tomkovic@gmail.com`
§`tánička@inmail`
¶`hmm`

3.1 Finger tree

3.2 Reversal tree

Táňa, čiň sa!

4 Haldy

4.1 d-nárna halda

4.2 Ľavicová halda

4.3 Skew halda

4.4 Párovacia halda

Katka, zase spíš?! [citacie]

5 Union-Find

Popis. Sú problémy, ktoré vyžadujú spájanie objektov do množín a množín navzájom a následné určovanie, do ktorej množiny objekt patrí. Od takejto *dátovej štruktúry pre disjunktné množiny* očakávame, že si bude udržiavať jednoznačného *zástupcu* každej množiny a bude poskytovať tieto tri oprácie:

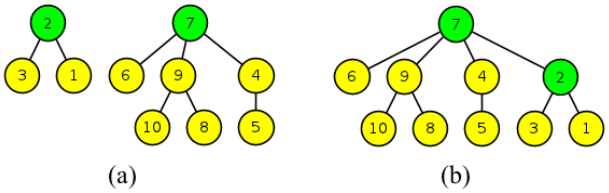
- **makeset**(x) – vytvorí novú množinu s jedným prvkom, ktorý nepatrí do žiadnej inej množiny;
- **find**(x) – nájde zástupcu množiny, v ktorej sa prvok x nachádza;
- **union**(x,y) – vytvorí novú množinu, ktorá obsahuje všetky prvky v množinách, ktorých zástupcovia sú x a y. Tieto množiny zmaže. Ďalej vyberie nového zástupcu novej množiny. Pre jednoduchosť, táto operácia predpokladá, že x a y sú zástupcovia množín.

Vďaka častej asociácií objektov a spájania množín ako vrcholy a hrany grafu sa často dátová štruktúra abstraktne reprezentuje ako *les* – množina zakorenených stromov. Konkrétnou implementáciou potom býva pole objektov – vrcholov. Ku každému objektu sa musí udržiavať smerník $p(x)$ na otca v strome. Smerník zástupcu množiny ukazuje na hodnotu `NULL`.

Operácia *makeset*(x) teda vytvorí nový prvok x a nastaví $p(x) =$ `NULL`.

Operáciu $find(x)$ vykonáme tak, že budeme sledovať cestu po smerníkoch, až kým nenájdeme zástupcu.

Operáciu *union*(x,y) ide najjednoduchšie vykonať tak, že presmerujeme smerník $p(y)$ na prvok x, teda $p(y) = x$. Môžeme ľahko pozorovať, že takýto *naivný* spôsob je neefektívny, lebo nám operácia $find(x)$ v najhoršom prípade, na n prvkoch, trvá $O(n)$ krokov.



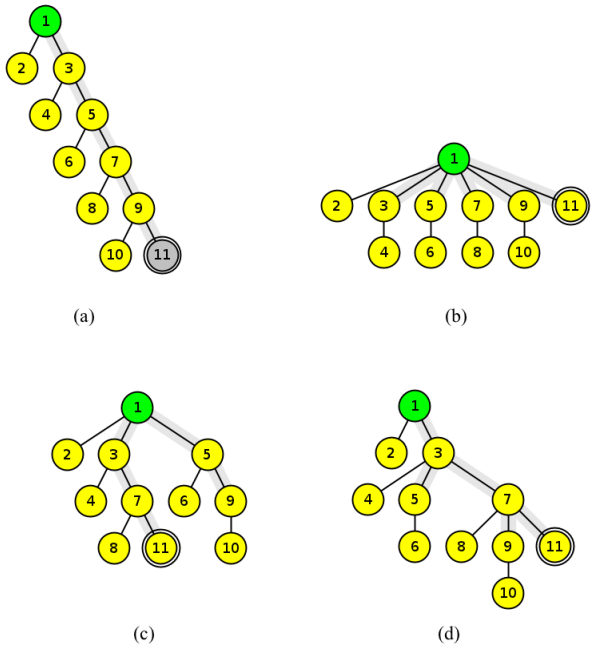
Obr. 1: *Spájanie podľa ranku.* (a) Pred spojením. (b) Po spojení. Plytší strom sa napojil pod hlbší.

Použitie. Vďaka dvom hlavným operáciam $find(x)$ a *union*(x,y) je táto dátová štruktúra známejšia pod pojmom *Union-Find*, ktorý používame aj my. Medzi najznámejšie problémy, ktoré sa riešia pomocou Union-Find patria Kruskalov algoritmus na nájdenie najlacnejšej kostry (Kruskal, 1956) a unifikácia (Knight, 1989). Veľmi priamočiare použitie je na zodpovedanie otázky „Koľko je komponentov súvislosti?“ alebo „Sú dva prvky v rovnakej množine?“ („Sú dva objekty navzájom prepojené?“), ak máme dovolené za behu pridávať hrany (spájať množiny objektov). Niektoré ďalšie grafové problémy popísal napr. Tarjan (1979).

Heuristika na spájanie. Existujú dva prístupy ako zlepšiť operácie a tým aj zrýchliť ich vykonanie. Sú to: heuristika *spájanie podľa ranku* a rôzne heuristiky na *kompresiu cesty*.

Prvá heuristika pridáva ku algoritmom hodnotu *rank*(x), ktorá bude určovať najväčšiu možnú hĺbku podstromu zakorenenú vrcholom x. V tom prípade pri operácií *makeset*(x) zadefinujeme *rank*$(x) = 0$. Pri operácií *union*(x,y) vždy porovnáme *rank*(x) a *rank*(y), aby sme zistili, ktorý zástupca predstavuje menší strom. Smerník tohto zástupcu potom napojíme na zástupcu s vyšším rankom. Zástupca novej množiny bude ten s vyšším rankom. Ak sú oba ranky rovnaké, vyberieme ľubovoľného zo zástupcov x a y, jeho rank zvýšime o jeden a smerník ostatného zástupcu bude ukazovať na tohto zástupcu. Zástupcom novej množiny bude vybratý zástupca.

Heuristiky na kompresiu cesty. Druhou heuristikou je kompresia cesty. Algoritmov na efektívnu kompresiu cesty je veľa (Tarjan and van Leeuwen,



Obr. 2: *Kompresie cesty.* (a) Pred vykonaním $find(x)$. (b) Jednoduchá kompresia. (c) Delenie cesty. (d) Pólenie cesty.

1984). Tu popíšeme tie najefektívnejšie. Prvou z nich je *jednoduchá kompresia cesty* (Hopcroft and Ullman, 1973). Pri vykonávaní operácie $find(x)$, po tom, ako nájdeme zástupcu množiny obsahujúcej prvok x, smerníky prvkov navštívených po ceste (včetne x) presmerujeme na zástupcu množiny. Toto síce spomalí prvé vykonávanie, ale výrazne zrýchli ďalšie hľadania. Druhou heuristikou je *delenie cesty* (Leeuwen and Weide, 1977). Pri vykonávaní operácie $find(x)$ pripojíme každý vrchol[1] v ceste od vrcholu x po koreň stromu na otca jeho otca. Treťou heuristikou je *pólenie cesty* (Leeuwen and Weide, 1977). Pri vykonávaní operácie $find(x)$ pripojíme každý druhý vrchol[2] v ceste od vrcholu x po koreň stromu na otca jeho otca.

Zdá sa, že najefektívnejšie je pólenie cesty pred delením, ktoré použije zhruba dvakrát viac priradení smerníkov a jednoduchou kompresiou, ktorá vyžaduje dva behy (Galil and Italiano, 1991).

[1]Okrem koreňa a synov koreňa, keďže tie deda a otca resp. deda nemajú.

[2]Okrem koreňa a synov koreňa, keďže tie deda a otca resp. deda nemajú.

6 Písmenkový strom

Abeceda je množina *znakov*. *Slovo* je postupnosť znakov z danej abecedy. *Písmenkový strom* je *zakorenený strom*, v ktorom každá hrana obsahuje práve jeden znak z abecedy alebo *ukončovací znak*. *Ukončovací znak* je ľubovoľný, dopredu dohodnutý symbol, ktorý sa v abecede nenachádza. My používame ako abecedu veľké znaky anglickej abecedy a ukončovací znak značíme dolárom (\$).

Popis. Strom obsahuje slová nazývané *kľúče*. Po prejdení cesty z koreňa do vrcholu s *ukončovacím znakom* prečítame kľúč.[3] Teda, oproti binárnym vyhľadávacím stromom je hlavný rozdiel v tom, že kľúče nie sú uložené vo vrcholoch, ale samotná poloha v strome určuje kľúč. Písmenkový strom sa podobne ako binárny vyhľadávací strom využíva ako *asociatívne pole (slovník)*, takže poskytuje tieto tri operácie:

- **insert**(k) – pridá do stromu kľúč k;
- **find**(k) – zistí, či sa v strom kľúč k nachádza;
- **delete**(k) – odstráni zo stromu kľúč k a prípadne vyrieši zmeny v strome.

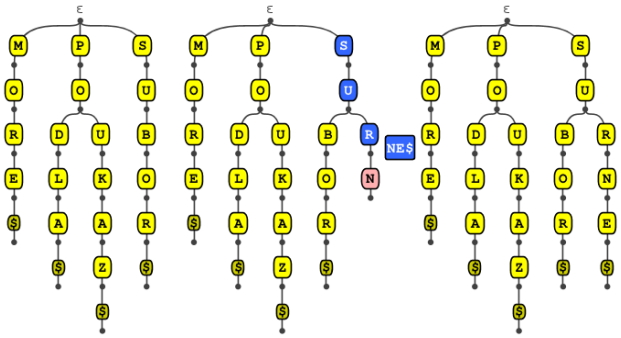
Všetky operácie začínajú v koreni a ku kľúču pridávajú ukončovací znak, teda pracujú s reťazcom $k\$$.

Operácia $insert(k)$ vloží do stromu vstupný reťazec tak, že z reťazca berie znaky a prechádza po príslušných hranách. Ak hrana so znakom neexistuje pridá ju.

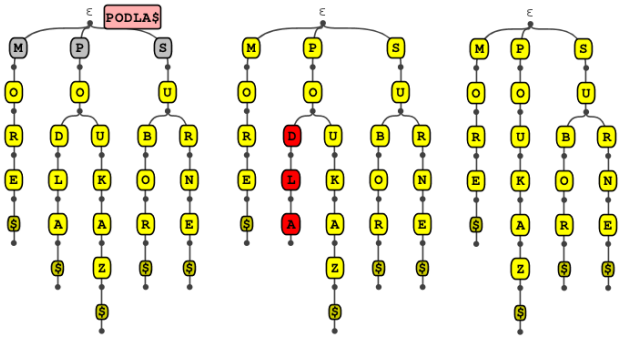
Operácia $find(k)$ sa spustí z koreňa podľa postupnosti znakov. Ak hrana, po ktorej sa má spustiť neexistuje, daný kľúč sa v strome nenachádza. Ak prečítame celý vstupný reťazec, daný kľúč sa v strome nachádza.

Operácia $delete(k)$ najprv pomocou operácie $find(k)$ zistí umiestnenie kľúča. Ak sa kľúč v strome nachádza, algoritmus odstráni hranu s ukončovacím symbolom a vrchol, ktorý bol na nej zavesený. V tomto štádiu sa nám môže stať, že v strome ostane takzvaná *mŕtva vetva* – nie je ukončená ukončovacím znakom. Pre fungovanie stromu to nevadí, všetky operácie by prebiehali správne, ale takto štruktúra zaberá zbytočne veľa miesta. Preto je dobré túto mŕtvu vetvu odstrániť.

[3]Na hranách $h_1, h_2, \ldots, h_n$ sú znaky $z_1, z_2, \ldots, z_{n-1}$,\$, ktoré po zreťazení utvoria slovo $z_1 z_2 \ldots z_{n-1}$.



Obr. 3: *Operácia insert*(k). Vo vizualizácií naznačujeme, kade sa presunieme.



Obr. 4: *Operácia delete*(k). *Mŕtva vetva* je vyznačená červenou.

Použitie. Vďaka tomu, že písmenkový strom udržiava spoločné *prefixy* kľúčov sa nazýva aj *prefixový strom*. Prvý krát popísal písmenkový strom Fredkin (1960), ktorý používal názov *trie memory*, keďže išlo o spôsob udržiavania dát v pamäti. Pojem *trie*[4] sa rozšíril a používa sa celosvetovo.

O niečo neskôr Knuth (1973) uviedol vo svojej knihe ako príklad na písmenkový strom vreckový slovník. V tom istom diele uviedol aj možnosť komprimovania vetiev a možnosť prerobenia *m*-árneho trie na binárny. Knuth (1973) však ukázal len komprimovanie koncov vetiev. Písmenkový strom, v ktorom každý vrchol, ktorého otec má len jedného syna je zlúčený s otcom[5], popísal Morrison (1968) a zaviedol pre ňho pojem *PATRICIA*. Pre túto dátovú štruktúru sa používa aj pojem *radix tree (radix trie)*.

Dátovú štruktúru podobnú *hashovacej tabuľke* a písmenkovému stromu popísal vo svojej dizertačnej práci Liang (1983) a nazval ju *packed trie*. Oproti obyčajnému písmenkovému stromu výrazne šetrila miesto. V práci ju využil na vytvorenie vzorcov pre slabikovanie slov.

Písmenkové stromy sa podobajú na *konečné automaty*. Vznikli rôzne modifikácie stromov na automaty, ktorých hlavnou výhodou je, že v komprimovanej podobe spájajú nielen predpony, ale aj prípony slov a teda v slovách ľudských jazykov výrazne znižujú pamäťový priestor potrebný na uchovanie dátovej štruktúry (Appel and Jacobson, 1988; Lucchesi et al., 1992).

Priamočiare je použitie písmenkového stromu na utriedenie poľa slov. Všetky slová sa pridajú do stromu a potom sa spraví *preorderový prechod* stromu. Túto myšlienku spracovali Sinha and Zobel (2004) a veľmi výrazne zrýchlil triedenie dlhých zoznamov slov. Neskôr tento algoritmus vylepšili Sinha et al. (2006). Kvôli tomu, ako algoritmus pracuje, sa nazýva *burstsort*.

Špeciálnym použitím písmenkového stromu je vytvorenie stromu zo všetkých prípon slova. Táto dátova štruktúra sa nazýva *sufixový strom* a dá sa modifikovať na udržiavanie viacerých slov. Tieto štruktúry majú veľmi veľa praktických využití (Gusfield, 1997).

7 História

ako sme ju do..
..robili.
Palyho umelecký opis.

8 Záver

work in progress; co sme spravili, preco sme lepsi, co este chceme/treba spravit, co je rozrobene Paly?

8.1 Príspevky autorov

Paly spravil to, Katka ono, Táňa zase chrastu a Friker si pospal pod stromom.

Poďakovanie

Autori by sa chceli poďakovať školiteľovi za veľa dobrých rád a odborné vedenie pri práci.

Literatúra

Aho, A. V. and Hopcroft, J. E. (1974). *The Design and Analysis of Computer Algorithms*. Addison-

[4] Z anglického re*trie*val – získanie.

[5] Na hranách teda nie sú znaky, ale slová.

Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Appel, A. and Jacobson, G. (1988). The world's fastest scrabble program. *Communications of the ACM*, 31(5):572–578.

Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.

Galil, Z. and Italiano, G. F. (1991). Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA.

Hopcroft, J. E. and Ullman, J. D. (1973). Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303.

Knight, K. (1989). Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124.

Knuth, D. E. (1973). *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

Kováč, J. (2007). Vyhľadávacie stromy a ich vizualizácia. Bakalárska práca.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50.

Leeuwen, J. and Weide, T. v. d. (1977). Alternative path compression rules. Technical report, University of Utrecht, The Netherlands. An outline of the results were presented at the Fachtagung on Algorithms and Complexity Theory, Oberwolfach, Oct 1977.

Liang, F. (1983). *Word hy-phen-a-tion by com-puter*. PhD thesis, Stanford University, Stanford, CA 94305.

Lucchesi, C. L., Lucchesi, C. L., and Kowaltowski, T. (1992). Applications of finite automata representing large vocabularies.

Morrison, D. R. (1968). Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534.

Saraiya, P., Shaffer, C. A., McCrickard, D. S., and North, C. (2004). Effective features of algorithm visualizations. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 382–386, New York, NY, USA. ACM.

Shaffer, C. A., Cooper, M. L., Alon, A. J. D., Akbar, M., Stewart, M., Ponce, S., and Edwards, S. H. (2010). Algorithm visualization: The state of the field. *Trans. Comput. Educ.*, 10:9:1–9:22.

Sinha, R., Ring, D., and Zobel, J. (2006). Cache-efficient string sorting using copying. *J. Exp. Algorithmics*, 11:1.2.

Sinha, R. and Zobel, J. (2004). Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9.

Tarjan, R. E. (1979). Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715.

Tarjan, R. E. and van Leeuwen, J. (1984). Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281.

Yao, A. C. (1985). On the expected performance of path compression algorithms. *SIAM J. Comput.*, 14:129–133.