

Gnarley Trees

Katka Kotrlová*

Pavol Lukča†

Viktor Tomkovič‡

Tatiana Tóthová§

Školiteľ: Jakub Kováč¶

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 algoritmov na týchto štruktúrach.

KLúčové 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 hálď to sú *d-nárna halda*, *l'avicová halda*, *skew halda* a *párovacia halda*. Tak tiež vizualizujeme aj *problém disjunktných 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 portáli <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 vizualizácie a implementované figúry čo 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

§taničkinmail

¶hmm

3.1 Finger tree

3.2 Reversal tree

Táňa, čiň sa!

4 Haldy

4.1 d -nárna halda

4.2 L'avicová halda

4.3 Skew halda

4.4 Párovacia halda

Katka, zase spíš?! [citácie]

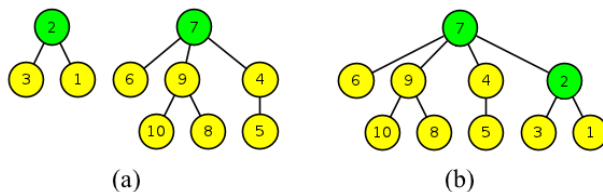
5 Union-Find

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 asociácii 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 $p(x) = \text{NULL}$. 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



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

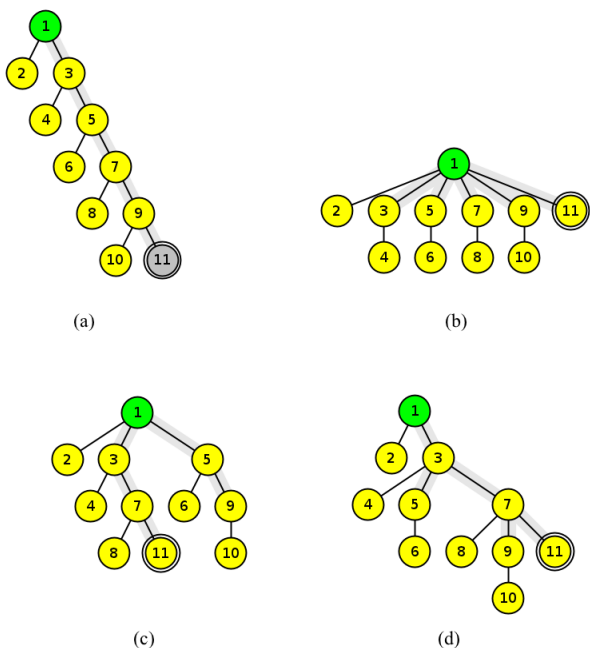
Ľahko pozorovať, že takýto *naivný* spôsob je neefektívny, lebo nám operácia **find**(x) v najhoršom prípade, na n prvkoch, trvá $O(n)$ krokov.

Použitie Vďaka dvom hlavným operáciám **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 sú popísané napr. v 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 zakorenenu vrcholom x . V tom prípade pri operácii **makeset**(x) zdefinujeme **rank**(x) = 0. Pri operácii **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 vybraný zástupca.

Heuristiky na kompresiu cesty Druhou heuristikou je kompresia cesty. Algoritmov na efektívnu kompresiu cesty je veľa (Tarjan and van Leeuwen, 1984). Tu popíšeme tie najefektívnejšie. Prvou z nich



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

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 (včetně 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 vrchol¹ 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 vrchol² 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).

¹okrem koreňa a synov koreňa, keďže tie deda a otca resp. deda nemajú

²okrem 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 sú vrcholy prázdne a každá hrana obsahuje práve jeden znak z abecedy alebo *ukončovací znak*. Strom obsahuje slová nazývané *kl'úče* a to tak, že kl'úč je rozdelený na znaky abecedy a tie sú umiestnené na jednotlivých hranách tak, že po prejdení cesty z koreňa do vrchola prečítame daný kl'úč.

Oproti binárnym vyhľadávacím stromom je hlavný rozdiel v tom, že *kl'úče* nie sú uložené vo vrcholoch, ale samotná poloha v strome určuje kl'úč. To, či je kl'úč v strome uložený určuje *ukončovací znak*. Môže to byť ľubovoľný symbol, ktorý sa nenachádza v abecede. My budeme používať ako abecedu veľké znaky anglickej abecedy a ukončovací znak značiť dolárom. (\$)

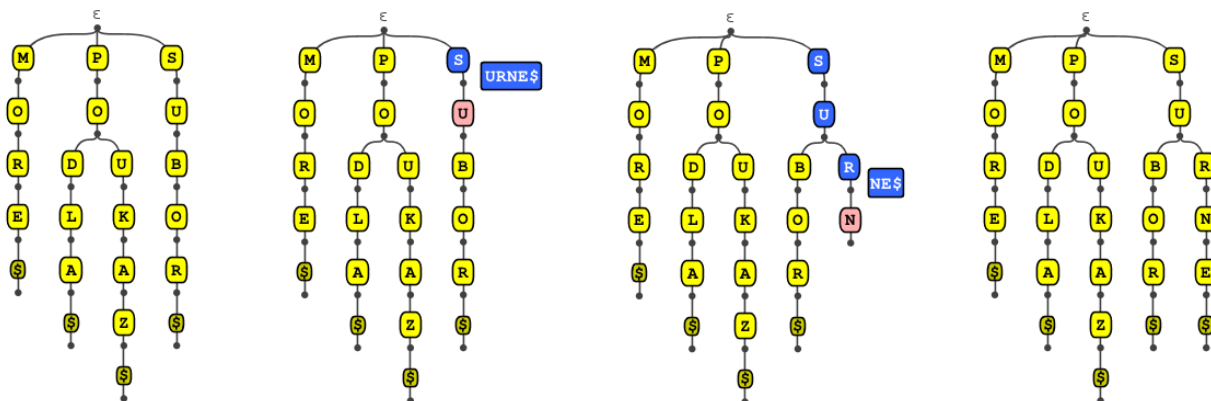
Popis Vzhľadom na svoj vznik sa písmenkový strom využíva ako *asociatívne pole (slovník)* a teda poskytuje tieto tri operácie:

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

Operácia $insert(k)$ začína v koreni. Zoberie kl'úč k a spraví z neho postupnosť znakov. Na koniec postupnosti zaradi ukončovací znak. Následne začne vykonávať cyklus, v ktorom vyberie z postupnosti ten znak, ktorý je na rade a presunie sa z momentálneho vrchola po príslušnej hrane. Ak hrana neexistuje pridá sa do stromu nový vrchol a príslušná hrana. Cyklus sknčí, keď sa vyčerpajú všetky znaky a teda sa do stromu vloží celý kl'úč a ukončovací znak.

Operácia $find(k)$ začína hľadať kl'úč k taktiež v koreni. Pridá ku postupnosti spravenej z kl'úča ukončovací znak (podobne ako pri $insert(k)$) a spustí sa z koreňa podľa postupnosti znakov. Ak hrana, po ktorej sa máme spustiť neexistuje, daný kl'úč sa v strome nenachádza. Ak prečítame celú postupnosť, daný kl'úč sa v strome nachádza.

Operácia $delete(k)$ najprv nájde hľadaný kl'úč pomocou operácie $find(k)$ a potom už len stačí odstrániť 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*



Obr. 3: Operácia $insert(k)$. Vo vizualizácii naznačujeme, kde sa presunieme.

– nie je ukončená ukončovacím znakom. Keďže si práve pamätáme smerník na mŕtvého syna môžeme ho odtrhnúť. A takto pokračovať vyššie až kým neutrhneme celú vetvu.

Použitie Písmenkový strom sa používa v rôznych modifikáciách a na rôzne účely. O zavedenie pojmov sa zaslúžil Fredkin (1960), ktorý používal operácie *storage* ($insert(k)$), *retrieval* ($find(k)$) a *deletion* ($delete(k)$) a dátovú štruktúru nazýval *trie memory*, keďže išlo naozaj o spôsob uloženia dát v pamäti. Výraz *trie*³ sa ujal a tak budeme písmenkový strom označovať aj v ďalšom texte. Vďaka tomu, že písmenkový strom udržiava spoločné *prefixy* klúčov sa nazýva aj *prefixový strom*.

O niečo neskôr Knuth (1973) uviedol vo svojej knihe ako príklad na trie vreckový slovník. V tom istom diele uviedol aj možnosť komprimovania vetiev a možnosť prerobenia *m*-árneho trie na binárny. Pre silnú podobnosť s *konečnými automatmi* vznikli prerobenie a implementácie trie pomocou komprimovaných automatov. Tieto majú použitie najmä ako pomôcky pri kontrole pravopisu (Lucchesi et al., 1992). Každý jazyk obsahuje veľa predpôn a prípon a ich kombinácií, preto takéto automaty zaberajú málo miesta.

Priamočiare je použitie trie na utriedenie poľa slov. Všetky slová sa pridávajú do stromu a potom sa spraví *preorderový prechod* trie. Túto myšlienku spracoval Sinha and Zobel (2004) a veľmi výrazne zrýchlil triedenie dlhých zoznamov slov. Neskôr tento algoritmus vylepšil Sinha et al. (2006). Špeciálnym použitím trie je zmena jeho prefixovej charakteris-

tiky na suffixovú a vytvorenie tzv. *suffixového stromu*. Táto štruktúra má 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.

Podakovanie

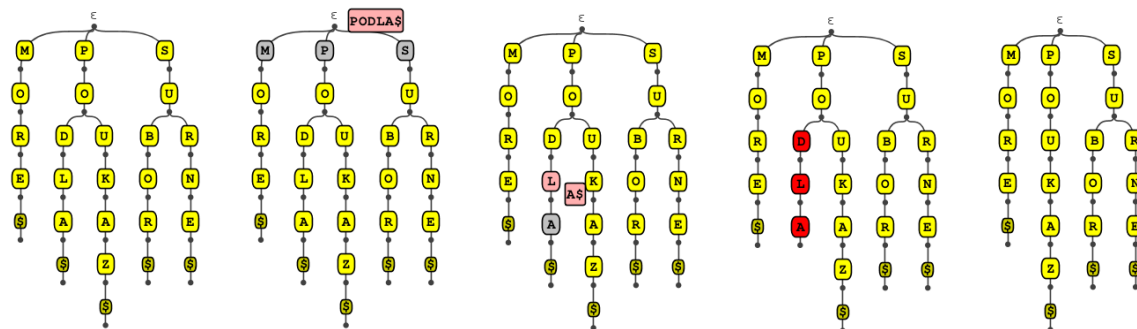
Autori by sa chceli poďakovať školiteľovi za veľa dobrých rád a odborné vedenie pri práci.

Literatúra

Aho, A. V. and Hopcroft, J. E. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

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

³Z anglického *retrieval* – získať.



Obr. 4: *Operácia delete(k)*. Najprv zistíme, či je kľúč v strome, potom ho zmažeme a potom zmažeme mŕtvu vetvu, ktorá je vyznačená červenou farbou.

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