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 algoritmoch na týchto štruktúrach.

Dostupnosť: Softvér je voľne dostupný na stránke `http://people.ksp.sk/~kuko/gnarley-trees`.

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

Projekt Gnarley Trees sme rozšírili nielen o vizualizácie ďalších dátových štruktúr, ale pribudli aj softvérové (vizualizačné) vylepšenia.

toto treba kompletne vymyslieť a napísať a potom tam dakde pichnúť:

Tesnejšie vykresľovanie grafov. V pôvodnej verzii programu sa stromy vykresľovali tak, že vertikálna súradnica predstavovala hĺbku v strome a horizontálna poradie vrcholu v *inorderovom prechode* stromu. Toto je ale spôsob, ktorý nešetrí priestor a

*algvis@googlegroups.com

pri štruktúrach ako písmenkový strom by výsledné stromy vyzerali škaredo. Preto sme sa rozhodli pre stromy implementovať alternatívny spôsob rozloženia, ktorý vymyslel **?** a pre n-árne stromy rozšíril **?**. Tieto rozloženia vykresľujú vrcholy stromov čo najtesnejšie, pričom dodržujú tieto pravidlá:

- vrcholy v rovnakej hĺbke sú vykreslené na jednej priamke a priamky určujúce jednotlivé úrovne sú rovnobežné;
- poradie synov je zachované;
- otec leží v strede nad najľavejším a najpravejším synom;
- izomorfné podstromy sa vykreslia identicky až na presunutie;
- ak vo všetkých vrcholoch vymeníme poradie všetkých synov, výsledný strom sa vykreslí zrkadlovo.

2.1 História krokov

Každá vizualizovaná operácia (insert/delete/...; ďalej len vizualizácia) na dátovej štruktúre pozostáva z niekoľkých krokov. Jednou z noviniek v projekte je možnosť vrátiť sa pri prehliadani operácií o niekoľko krokov späť (história krokov), resp. vrátiť späť celé operácie (história operácií).

Niekedy sa stáva, že nedočkavý užívateľ rýchlo prekliká cez celú vizualizáciu a pritom si nestihne uvedomiť, aké zmeny sa vykonali na danej dátovej štruktúre. Inokedy je operácia taká rozsiahla, že niektoré dôležité zmeny si nevšimne. Vtedy by bolo užitočné pozrieť si vizualizáciu ešte raz (alebo niekoľkokrát). Tento problém rieši história krokov. Užívateľ má možnosť vrátiť sa späť o jeden krok (tlačidlo „Späť“/„Previous“) alebo preskočiť na ľubovoľný krok po kliknutí na zodpovedajúci komentár.

História krokov a operácií je atomická. Krok/operácia sa vykoná/vráti celý(-á) alebo vôbec, pričom stav dátovej štruktúry korešponduje s pozíciou v histórii. To umožňuje po vrátení celej operácie vykonať inú operáciu. Táto vlastnosť je užitočná najmä v prípade vykonania operácie (prípadne zmazania celej dátovej štruktúry) omylom.

2.2 Ďalšie rozšírenia

Patrí k nim možnosť priblíženia/vzdialenia a presunu vykreslenej dátovej štruktúry v rámci vizualizačnej plochy. Užívateľ túto funkcionalitu využije najmä pri dátových štruktúrach s veľkým počtom prvkov, kedy je obmedzený veľkosťou plochy. Ďalším rozšírením je výpis celej postupnosti komentárov vizualizovanej operácie, ktorý prináša spolu s históriou krokov značné zjednodušenie výučby. Užívateľ si môže konkrétnu vizualizáciu pozrieť toľko krát, koľko potrebuje na jej správne pochopenie. Navyše vidí, aké kroky budú nasledovať/predchádzať a podľa toho si môže určiť vlastné tempo prezerania vizualizácie. Ak si myslí, že daným krokom už porozumel, môže ich preskočiť.

3 Vyvážené stromy

Vyvážený strom je taký strom, kde pre každý vrchol platí, že rozdiel hĺbok jeho ľavého a pravého podstromu je najviac 1. V dobre vyváženom strome je rozdiel hĺbok rovný nule.

3.1 B+-strom

Popis. *B+-strom* je variácia B stromu taká, že má všetky kľúče uložené v listoch a všetky vrcholy, ktoré sú listami, sú spojené do spájaného zoznamu. B strom je zakorenený utriedený strom, ktorý má rád n (v jednom vrchole maximálne $n-1$ prvkov) a limity na maximálny (n) a minimálny ($\lfloor \frac{n}{2} \rfloor$) počet potomkov. Vďaka tomu je dobre vyvážený a jeho operácie sú vykonávané v logaritmickom čase. Vzhľadom na usporiadanie budem predpokladať, že menšie prvky sa nachádzajú vždy vľavo. *B+-strom* je *asociatívne pole (slovník)*, čiže poskytuje tieto tri operácie:

- $insert(a)$ – pridá do stromu a;
- $find(a)$ – zistí, či sa v strome nachádza a;
- $delete(a)$ – odstráni zo stromu a.

Operácia $find\,(a)$ začne v koreni, nájde v ňom prvý kľúč väčší od hľadaného. Nech je i-ty v poradí, potom hľadanie pokračuje v i-tom synovi tohto vrchola. Je zrejmé, že ak sa väčší kľúč nenájde, presunieme sa do posledného n-tého syna. V liste sa už len skontroluje, či sa v ňom hľadaný kľúč nachádza.

Definujem dve operácie: COPY-UP a PUSH-UP, ktoré používa operácia $insert\,(a)$Ak má vrchol viac prvkov, ako je maximálny limit, treba ho zmenšiť. Rozdelí sa na dve časti. Ak vrchol nie je listom, použije sa PUSH-UP, najmenší kľúč pravej časti sa vyberie a stane sa otcom vytvorených dvoch častí. Pokiaľ

to list je, kľúč v ňom musí zostať, preto sa iba skopíruje. Táto operácia sa nazýva COPY-UP.

Operácia *insert*(a) najprv pomocou operácie $find(a)$ zistí, či štruktúra daný kľúč obsahuje. Ak nie, je zrejmé, že patrí práve do vrchola, kde $find(a)$ skončil. Ak vrchol má po vložení viac kľúčov ako maximálny limit, je treba ho zmenšiť. Pokiaľ otcovský vrchol má syna, do ktorého je možné kľúč presunúť a zároveň syn susedí s veľkým vrcholom, postup je nasledovný. Nech je vrchol vľavo menší a terajší vrchol je i-ty syn v poradí. Potom algoritmus z neho vyberie najmenší kľúč, presunie ho na miesto $(i-1)$-ho kľúča v otcovskom vrchole. Vymenený kľúč následne vloží do $(i-1)$-ho syna. Ak sú susedné vrcholy príliš veľké na presun kľúča, použije sa operácia COPY-UP. Nový vrchol s jedným kľúčom, ktorý vznikol, vložíme do otcovského vrchola. Ak otcovský vrchol presiahol najväčšiu možnú veľkosť, znova sa aplikuje popísaný algoritmus s jedným rozdielom - namiesto COPY-UP sa použije PUSH-UP.

Operácia $delete(a)$ najprv pomocou $find(a)$ nájde kľúč, potom ho z vrcholu odstráni. Tento vrchol môže mať po odstránení menší počet kľúčov ako minimálny limit. Vtedy, ak sa dá, sa prenesie jeden kľúč zo súrodenca. Ak sa nedá, vrchol sa s ním zlúči. Zároveň sa k nim pridá aj kľúč z otcovského vrchola, ktorý ich rozdeľoval. Pokiaľ to spôsobilo, že otcovský vrchol má menej kľúčov, ako je povolené, znova sa aplikuje predošlý algoritmus. Keďže na koreň sa nevzťahuje minimálny limit, po skončení bude strom zaručene v konzistentnom tvare.

Časová zložitosť Vkladanie, vymazávanie a hľadanie má časovú zložitosť $O(\log_B(n))$**?**. Kde B je veľkosť najmenšej stránky záznamov a n je počet prvkov.

Použitie. B+-strom podporuje efektívne vyhľadanie prvkov poľa, ktoré patria do daného intervalu**?**. Algoritmus nájde jeden krajný bod a vďaka spájanému zoznamu, vytvorenému z listov, ostatné prvky postupne prečíta. Zložitosť je $O(\log_B(n) + \frac{t}{B})$, kde t je počet výsledných kľúčov z hľadaného intervalu. Keď sa pozrieme na širšie využitie B+-stromu, zistíme, že je použitý takmer vo všetkých databázových systémoch. Ak zvolíme vhodný rád, vieme jednotlivé vrcholy dobre napasovať na stránky a tým regulovať ako počet prístupov k pamäti, tak jej zaplnenie**?**. Pridáva mu aj fakt, že pri implementácii agregačných funkcií na túto štruktúru vychádza ich zložitosť na $O(\log_B(n))$. Ďalšia výhoda, ktorú táto štruktúra poskytuje, sa prejaví, ak máme dáta a chceme ich vložiť do štruktúry. V iných prípadoch by bolo treba spraviť n vložení. Avšak spájaný zoznam na najnižšej vrstve umožní vystavať B+-strom odpodu. Takýto postup nám zaručí zložitosť $O(\frac{n}{B} * \log_B(n))$, čo je B-krát rýchlejšie ako tradičný spôsob.

3.2 Strom s prstom

Strom s prstom (z anglického Finger tree) je vyhľadávací strom, kde prst je smerník na nejaký vrchol. Jeho poloha sa využíva pri všetkých operáciách. Vďaka tomu má strom s prstom efektívnejšie vyhľadávanie, vkladanie a mazanie kľúčov, ktoré sú v blízkom okolí prsta.

Popis. Strom s prstom je upravený (2,4)-strom. (2,4)-strom je dobre vyvážený utriedený zakorenený strom, kde všetky listy majú rovnakú hĺbku a vnútorné vrcholy majú stupeň 2, 3 alebo 4. Kľúče sú uložené v listoch a vnútorné vrcholy obsahujú ich kópie, aby viedli vyhľadávanie. Pre podporu vyhľadávania s prstom bola táto štruktúra rozšírená o hrany na vrstvách, čo znamená, že všetky vrcholy na rovnakej vrstve (také, ktoré sú rovnako vzdialené od koreňa) sú pospájané do obojsmerného spájaného zoznamu. Ak sú nejaké dva vrcholy spojené takouto hranou, budeme hovoriť, že sú susedia. Prst, ako už bolo spomenuté, ukazuje na nejaký vrchol. Môže sa pohybovať po všetkých hranách a pomocou neho sa vykonávajú všetky operácie. Keďže sú všetky kľúče uložené v listoch, prst na tejto vrstve začína, aj končí. Vzhľadom na usporiadanie budem predpokladať, že menšie prvky sa nachádzajú vždy vľavo a každý prekopírovaný kľúč má svoj originál vždy vľavo v listovom vrchole. Strom s prstom je *asociatívne pole (slovník)*, čiže poskytuje tieto tri operácie:

- **insert**(a) – pridá do stromu a;
- **find**(a) – zistí, či sa v strome nachádza a;
- **delete**(a) – odstráni zo stromu a.

Operácia $find(a)$ začne na mieste, kam ukazuje prst. Skontroluje, či by kľúč mal patriť do daného vrchola. Ak nie, pozrie sa, či nepatrí do niektorého zo susedov. Ak áno, prst sa tam presunie a vyhľadávanie sa skončilo. Inak smerník prejde o vrstvu vyššie, na otcovský vrchol. Ak hľadaný kľúč patrí do jeho podstromu (t.j. je väčší ako jeho najmenší kľúč a menší ako ten najväčší), zíde po hranách do listu, kde by sa

daný kľúč mal nachádzať. Keď do podstromu nepatrí, skontroluje, či nepatrí do podstromu susedov. Ak áno, prejde po vrstevnej hrane na suseda a následne zíde až do listu, kde by mal kľúč byť. Pokiaľ prst nenarazil na správny podstrom, znova sa presunie smerom nahor po otcovskej hrane. Hľadanie pokračuje analogicky. Je zrejmé, že ak prst ukazuje na koreň, kľúč bude patriť do jeho podstromu.

Otázka patričnosti do podstromu sa dá pre krajné prípady optimalizovať. Ak totiž vrchol, na ktorý ukazujeme, nemá napríklad pravého suseda, je zrejmé, že väčší kľúč ako je najväčší v tomto vrchole v strome nie je. Preto ľubovoľný kľúč do podstromu tohto vrchola patrí práve vtedy, keď je väčší ako jeho najmenší prvok. Analogicky to platí, ak chýba ľavý sused.

Operácia *insert*(a) najprv pomocou operácie $find(a)$ nájde miesto, kam by mal vkladaný kľúč patriť. Ak taký kľúč už v strome je, ďalší sa nevloží. V prípade, že sme vložili nový kľúč, môže sa stať, že vrchol „pretečie“, tzn. má viac ako 3 prvky. Situácia sa vyrieši rovnako ako v B+-strome. Operácia $delete(a)$ najprv pomocou operácie $find(a)$ nájde miesto, kam by mal hľadaný kľúč patriť. Ak tam nie je, vymazávanie sa končí. Inak sa vymaže. Môže sa stať, že vrchol „podtečie“, tzn. nemá kľúč. Tento problém sa rieši rovnako ako v B+-strome.

Časová zložitosť. Keďže každý vrchol má stupeň minimálne 2, (2,4)-strom má hĺbku $O(\log(n))$, kde n je počet kľúčov, a teda podporuje vykonávanie operácií v čase $O(\log(n))$. Ak sa však použije prst, časová zložitosť vychádza na $O(\log(d))$, kde d je vzdialenosť pozície prsta a vrcholu, kam patrí cieľový kľúč, amortizovane dokonca na $O(1)$**?**.

Použitie. Prst ako smerník na prvok štruktúry, ktorý umožňuje efektívnejší prístup k okolitým kľúčom, prvýkrát spomenuli Guibas et al. Vo svojej publikácii prezentujú B-strom podporujúci vyhľadávanie v $O(\log(n))$ a update-y dokonca v $O(1)$ čase, predpokladajúc, že je udržiavaných len $O(1)$ pohyblivých prstov**?**. Pohyb prsta o d pozícií trvalo $O(\log(n))$ času. Na základe tejto práce navrhli Huddleston a Mehlhorn svoj vrstvovo spájaný (2,4)-strom, ktorý bol neskôr upravený vďaka Belloch et al. na priestorovo efektívnejšiu alternatívu. Toto riešenie využíva jeden prst, s ktorým štruktúra ponúka rovnakú operačnú zložitosť ako (2,4)-stromy. Model bol dokonca zovšeobecnený na (a,b)-stromy, kde $b \geq 2a$. Je zaujímavé, že pre (2,3)-strom, ktorý nespĺňa danú podmienku, bola nájdená postupnosť vkladaní a vymazávaní taká, že vyžadovala až $O(n\log(n))$ čas**?**.

Vizualizácia. Strom s prstom je vizualizovaný pomocou B+-stromu s rádom 4, keďže jeho podmienky pre počet potomkov vyhovujú danej štruktúre. Prst je samostaný pohyblivý článok, ktorý si pamätá iba vrchol, na ktorý ukazuje. Po stromovitej štruktúre sa vie hýbať vďaka informáciám získaným z daného vrcholu.

3.3 Strom s reverzami

Strom s reverzami (z anglického Reversal tree) je dátová štruktúra na uchovávanie permutácií. Má stromovitý charakter, teda poradie prvkov môžeme chápať rovnako ako v usporiadaných stromoch (t.j. najľavejší prvok je na prvom mieste).

Popis. Strom s reverzami je založený na Splay stromeKováč (2007). Splay strom je štruktúrou binárny strom, líši sa od neho iba operáciami. Keď pracuje s ľubovoľným prvkom, na konci operácie bude vo vrchole buď daný kľúč alebo najbližší z jeho okolia (vďaka splay algoritmu - preto ten názov). Na rozdiel od Splay stromu, strom s reverzami nepracuje s kľúčmi, ale s poradím prvkov. Preto je nutné, aby mal každý vrchol zaznamenané, aký veľký je jeho podstrom (vrátane neho). Ďalšiou informáciou, o ktorú sú vrcholy rozšírené, je vlajka, ktorú vrchol môže a nemusí mať. Ak ju má, potom podstrom daného vrchola má byť čítaný reverzne.

Strom s reverzami poskytuje tieto tri operácie:

- **insert**(a) – pridá do stromu a;
- **find**(a) – zistí, ktorý prvok je na a-tom mieste;
- **reverse**(a,b) – reverzne interval od a po b.

Operácia $find(a)$ pomocou veľkosti podstromov zistí, ktorý prvok je na požadovanom mieste. Po nájdení pozície sa použije splay algoritmus, aby hľadaný prvok skončil v koreni. Ak niekedy narazí na vrchol s vlajkou, použije algoritmus na odstránenie vlajky, ktorý vyzerá nasledovne. Danému vrcholu odstráni vlajku, vymení mu synov a každému z nich zmení status vlajky.

Operácia *insert*(a) najprv pomocou operácie $find(a)$ dostane do koreňa posledný prvok. Je zrejmé,

že na mieste pravého syna nie je žiaden vrchol. Následne sa na toto miesto pripojí desať prvkov už prepojených v stromovitej štruktúre.

Operácia *reverse*(a,b) pomocou operácie *find*(a) dostane prvok na pozícii vľavo od intervalu na miesto koreňa a vymaže hranu medzi koreňom a pravým synom. To spôsobí, že vzniknú dva stromy T1 a T2, pričom v T1 budú všetky prvky, ktoré sú pred intervalom. Následne použije *find*(a) v strome T2 na prvok vpravo od intervalu, a aj ten oddelí tak, že zruší hranu medzi koreňom a jeho ľavým synom. Tým vznikli T3 a T4, kde T4 obsahuje prvky za intevalom. Zostane strom T3 - interval, ktorý chceme reverznúť, takže koreňu zmeníme status vlajky. Následne pospájame strom dohromady. T3 bude ľavý syn T4 a T4 bude pravý syn T1.

Všetky operácie splay tree sú amortizovane $O(\log(n))$. Nakoľko samotné reverzovanie je v$O(\log(1))$, časová zložitosť operácií v strome s reverzami bude tiež amortizovane $O(\log(n))$.

Použitie. Dátových štruktúr na ukladanie permutácií je mnoho. Splay strom ako základ si vzali Fredman et al., pretože ponúkali dobré konštanty a zároveň umožňovali elegantnú implementáciu rozdeľovania a spájania postromov**?**.

Vizualizácia. Pre lepšiu vizualizáciu sme pridali do stromu nultý a posledný prvok. Tieto prvky do reverzovateľného intervalu nepatria, majú však zmysel v prípade, ak sa reverzuje interval, ktorý zahŕňa aspoň jeden okraj. V tom prípade v operácii *reverse*(a,b) nezostane ani T1 ani T4 prázdny. Aby nevznikli problémy s operáciami, za krajné kľúče boli zvolené hodnoty 0 a číslo o jedna väčšie od aktuálneho maxima. Zároveň, pre lepšiu predstavu, bolo pridané pole, v ktorom užívateľ vidí skutočné poradie prvkov, ktoré zo stromu nie je až tak zjavné. Pole simuluje operácie spolu so stromom, ale tie sú na ňom vykonávané v lineárnom čase.

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. V niektorých aplikáciach potrebujeme udržiavať prvky rozdelené do skupín (disjunktných množín), pričom skupiny sa môžu zlučovať a my potrebujeme pre daný prvok efektívne zistiť, do ktorej skupiny patrí. Predpokladáme, že každá množina S je jednoznačne určená jedným svojim zástupcom $x \in S$ a potrebujeme implementovať nasledovné tri operácie:

- *makeset*(x) – vytvorí novú množinu $S = \{x\}$ s jedným prvkom;
- *union*(x,y) – ak x,y sú zástupcovia množín S a T, *union* vytvorí novú množinu $S \cup T$, pričom S aj T zmaže. Zástupcom novej množiny $S \cup T$ je x alebo y.
- *find*(x) – nájde zástupcu množiny, v ktorej sa prvok x nachádza.

V takejto situácií je vhodná štruktúra *union-find*. Táto dátová štruktúra sa reprezentuje ako les, kde každý strom zodpovedá jednej množine a korene stromov sú zástupcovia množín. Pri implementácií si stačí pre každý prvok x udržiavať smerník $p(x)$ na jeho otca (pre koreň je $p(x) = \texttt{NULL}$).

Operácia *makeset*(x) teda vytvorí nový prvok x a nastaví $p(x) = \texttt{NULL}$.

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.

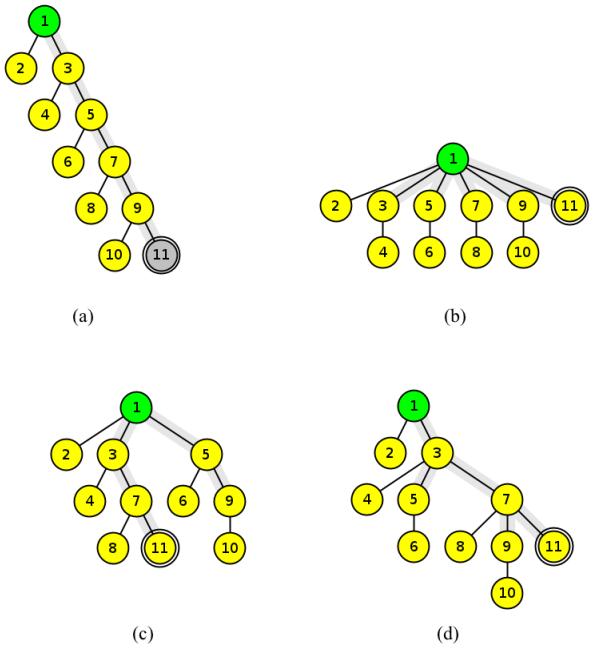
Použitie. Medzi najznámejšie problémy, ktoré sa riešia pomocou union-find patria Kruskalov algoritmus na nájdenie najlacnejšej kostry (Kruskal, 1956) a unifikácia (Knight, 1989). Veľmi priamočiare použitie je na zodpovedanie otázky „Koľko je komponentov súvislosti?“ alebo „Sú dva prvky v rovnakej množine?“ („Sú dva objekty navzájom prepojené?“), ak máme dovolené za behu pridávať hrany (spájať množiny objektov). Niektoré ďalšie grafové problémy popísal napr. Tarjan (1979).

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

Heuristika na spájanie. 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, 1984). Tu popíšeme tie najefektívnejšie. Prvou z nich je *úplná kompresia* (Hopcroft and Ullman, 1973). Pri vykonávaní operácie $find(x)$, po tom, ako nájdeme zástupcu množiny obsahujúcej prvok x, smerníky prvkov navštívených po ceste (vrátane 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 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 v ceste od vrcholu x po koreň stromu na otca jeho otca.

Časová zložitosť union-findu záleží od toho, koľko prvkov je v množinách a koľko je operácií celkovo vykonaných operácií. Všetky uvedené spôsoby ako



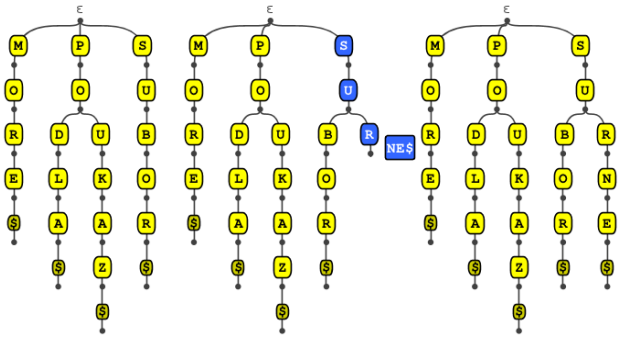
Obr. 1: *Kompresia cesty z vrcholu 11 do koreňa.* Cesta je vyznačená šedou. (a) Pred vykonaním kompresie. Pri úplnej kompresii (b) sa všetky vrcholy napoja na zástupcu. Pri delení cesty (c) a pólení cesty (d) sa cesta skráti približne na polovicu.

vykonať operáciu $find(x)$ sa dajú použiť s obomi realizáciami operácie $union(x,y)$. Počet prvkov označme n a počet operácií m. V praxi je zvyčajne počet operácií oveľa väčší ako počet prvkov. Pri tomto predpoklade ($m \geq n$) je pri použití spájania podľa ranku časová zložitosť pre algoritmus bez kompresie $\Theta(m \log n)$ a pre všetky tri uvedené typy kompresií $\Theta(m\,\alpha(m,n))$ (Tarjan and van Leeuwen, 1984).

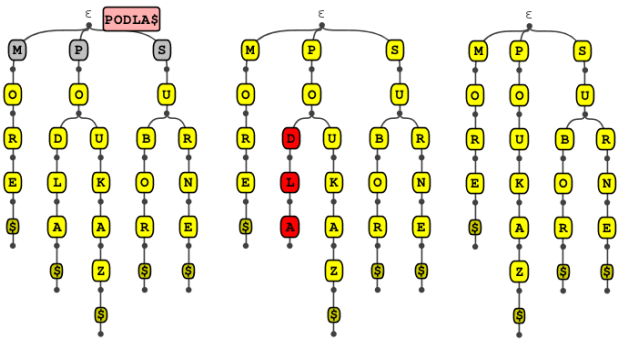
Vizualizácia. Union-find sme vizualizovali ako les. Pre názorné oddelenie množín sme si zvolili pravidlo, ktoré zakazovalo vykresliť vrchol napravo od najľavejšieho vrcholu a naľavo od napravejšieho vrcholu inej množiny. Jednotlivé množiny sme už vykreslovali tesným Walkerovým algoritmom (**?**). Vizualizácie poskytuje všetky vyššie spomínané heuristiky a aj tlačidlo na vykonanie viacerých náhodných spojení naraz. Toto je užitočné, keď chce užívateľ vidieť, ako dátová štruktúra vyzerá, po vykonaní veľa operácií.

6 Písmenkový strom

Písmenkový strom reprezentuje množinu slov. Oproti binárnym vyhľadávacím stromom je hlavný rozdiel



Obr. 2: *Vloženie slova „*SURNE*“*. Začiatok slova „SU“ sa v strome nachádza, ešte treba pripojiť hrany so znakmi R, N, E a $.



Obr. 3: *Odstránenie slova „*PODLA*“*. Po odstránení $ nám v strome ostane nepotrebá prípona „DLA“ (mŕtva vetva), ktorá je vyznačená červenou.

v tom, že kľúče nie sú uložené vo vrcholoch, ale samotná poloha v strome určuje kľúč (slovo).

Popis. Písmenkový strom je *zakorenený strom*, v ktorom každá hrana obsahuje práve jeden znak z abecedy alebo *ukončovací znak*. Teda, každá cesta z koreňa do listu so znakmi $w_1, w_2, \ldots, w_n, \$$ prirodzene zodpovedá slovu $w = w_1 w_2 \cdots w_n$. *Ukončovací znak* je ľubovoľný, dopredu dohodnutý symbol, ktorý sa v abecede nenachádza (napr. $).

Písmenkový strom je *asociatívne pole (slovník)*, čiže poskytuje tieto tri operácie:

- *insert*(w) – pridá do stromu slovo w;
- *find*(w) – zistí, či sa v strome slovo w nachádza;
- *delete*(w) – odstráni zo stromu slovo w.

Všetky operácie začínajú v koreni a ku slovu pridávajú ukončovací znak, teda pracujú s reťazcom $w$$.

Operácia *insert*(w) vloží do stromu vstupný reťazec tak, že z reťazca číta znaky a prechádza po príslušných hranách. Ak hrana s daným symbolom neexistuje, pridá ju (pozri obr. 2).

Operácia *find*(w) sa spustí z koreňa podľa postupnosti znakov. Ak hrana, po ktorej sa má spustiť neexistuje, dané slovo sa v strome nenachádza. Ak prečíta celý vstupný reťazec, dané slovo sa v strome nachádza.

Operácia *delete*(w) najprv pomocou operácie *find*(w) zistí umiestnenie slova. Ak sa slovo v strome nachádza, algoritmus odstráni hranu s ukončovacím znakom a vrchol, ktorý bol na nej zavesený. V tomto štádiu sa nám môže stať, že v strome ostane *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 (pozri obr. 3).

Všetky tri operácie majú časovú zložitosť $O(|w|)$, kde $|w|$ je dĺžka slova.

Použitie. Prvýkrát navrhol písmenkový strom Fredkin (1960), ktorý používal názov *trie memory*[1] *insert* (w)ý [1]Z anglického re*trie*val – získanie., keďže išlo o spôsob udržiavania dát v pamäti.

O niečo neskôr Knuth (1973) uviedol vo svojej knihe ako príklad na písmenkový strom vreckový slovník. Knuth (1973) však ukázal len komprimovanie koncov vetiev. Písmenkový strom, v ktorom každý vrchol, ktorého otec má len jedného syna je zlúčený s otcom[2] *insert* (w)ý [2]Na hranách teda nie sú znaky, ale slová., vymyslel Morrison (1968) a zaviedol pre ňho pojem *PATRICIA* (*radix tree, resp. radix trie*). Využíva sa napríklad v *routovacích tabuľkách* (Sklower, 1991).

Pôvodný návrh (Fredkin, 1960) ako uložiť trie do pamäte zaberal príliš veľa nevyužitého priestoru. Liang (1983) navrhol ako efektívne zmenšiť pamäťový priestor potrebný na uloženie trie. Nazval ho *packed trie* a navrhol spôsob, akým ho dobre použiť na slabikovanie slov. Systém bol následne použitý v programe TEX.

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. Vďaka tomu sa využívajú na jazykovú korekciu,

automatické dopĺňanie slov a podobne (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).

Vizualizácia. Pri vizualizácií písmenkového stromu sme použili Walkerov algoritumus pre úsporné rozloženie vrcholov v strome. (**?**) Keď má vrchol viacej synov a hrany kreslíme priamo, tak vzniká nedostatok priestoru pre umiestnenie znakov na hrany. Preto sme sa rozhodli kresliť hrany zakrivene, podľa Bézierovej krivky určenej štyrmi bodmi. Vo vizualizácií sa dajú vložiť náhodné slová podľa momentálne nastaveného jazyka. Taktiež sa automaticky odstraňuje diakritika a interpunkcia, takže sa dá naraz vložiť súvislý text.

7 Záver

work in progress; co sme spravili - uz je v uvode?, preco sme lepsi - sme vobec lepsi (ako galles)?, co este chceme/treba spravit - maybe DONE, co je rozrobene - v podstate to, co chceme spravit - same as previous?
V ďalšej práci sa budeme zaoberať implementovaním ďalších vizualizácií dátových štruktúr (napríklad TODO „tejto, tejto a aj tejto“), doplnením histórie krokov do všetkých dátových štruktúr, ale aj vylepšením GUI, refaktorovaním zdrojového kódu a inými softvérovými vylepšeniami. Našim cieľom je čo najviac zjednodušiť prácu s programom a tak zefektívniť výučbu jednotlivých dátových štruktúr, resp. spraviť ju zábavnejšou.

7.1 Príspevky autorov

Katka Kotrlová obohatila projekt o vizualizácie d-nárnej, ľavicovej, skew a párovacej haldy, Viktor Tomkovič pridal vizualizácie union-findu a písmenkového stromu, Tatiana Tóthová vizualizovala finger tree a Pavol Lukča dorobil históriu krokov a operácií do takmer všetkých slovníkov a venoval sa refaktorovaniu zdrojového kódu. Na príprave tohto textu sa podieľali všetci autori.

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-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.

Patwary, M., Blair, J., and Manne, F. (2010). Experiments on union-find algorithms for the disjoint-set data structure. *Experimental Algorithms*, pages 411–423.

Reingold, E. and Tilford, J. (1981). Tidier drawings of trees. *Software Engineering, IEEE Transactions on*, (2):223–228.

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.

Sklower, K. (1991). A tree-based packet routing table for berkeley unix. In *Proceedings of the Winter 1991 USENIX Conference*, pages 93–104.

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.

Walker II, J. Q. (1990). A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20(7):685–705.

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