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

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 pri štruktúrach ako písmenkový strom by výsledné stromy vyzerali škaredo. Preto sme sa rozhodli pre

*`algvis@googlegroups.com`

stromy implementovať alternatívny spôsob rozloženia, ktorý vymyslel Reingold and Tilford (1981) a pre n-árne stromy rozšíril Walker II (1990). 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.

3 Vyvážené stromy

uz boli, pribudli

3.1 Finger tree

3.2 Reversal tree

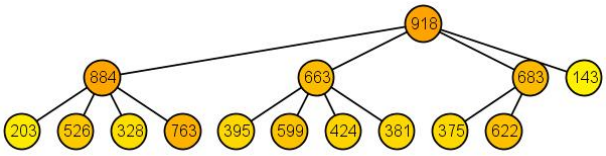
Táňa, čiň sa!

4 Haldy

V nasledujúcom texte sa budeme zaoberať rôznymi druhmi prioritných front. Popíšeme *d-árnu haldu* ako základnú modifikáciu binárnej haldy, *Ľavicovú haldu* a niektoré druhy samoupravujúcich sa háld, konkrétne *Skew haldu* a *Párovaciu haldu*. Halda je vo všeobecnosti *zakorenený strom* s vrcholmi obsahujúcimi kľúče reprezentujúce dáta. Dôležitá je zakladná podmienka haldy, ak vrchol$p(x)$ je otcom vrcholu x, potom $kľúč(p(x)) \leq kľúč(x)$[1]. Štandardné operácie, ktoré haldy podporujú, a ktorými sa budeme zaoberať pri každej dátovej štuktúre, sú:

- **createHeap** – vytvorí prázdnu haldu;
- **insert** (x) – vloží vrchol s kľúčom x;

[1] Bez ujmy na všeobecnosti budeme uvažovať o *min haldách*, teda v koreni sa bude nachádzať najmenší prvok. Podobnými úvahami by sme text mohli rozšíriť o *max haldy* s najväčším prvkom v koreni.



Obr. 1: Príklad d-árnej haldy pre $d = 4$

- **findMin** – vráti minimum t.j. hodnotu kľúča v koreni;
- **deleteMin** – odstráni vrchol s najmenším kľúčom, t.j. koreň;
- **decreaseKey** (v, Δ) – zníži kľúč vrcholu v o delta;

Niektoré haldy navyše implementujú **meld** (i, j) – spojí haldu i s haldou j.

4.1 d-árna halda

Popis. Ako zovšeobecnenie binárnej haldy môžeme považovať d-árnu haldu. Rozdiel je v stupni vrcholov. D-árna halda je, až na poslednú úroveň, *úplný d-árny strom* spĺňajúci podmienku haldy. Posledná úroveň je *zľava úplná*. Halda sa najčastejšie reprezentuje v poli, koreň je na mieste 0 a synovia i-teho prvku sú v poli na miestach $(d \cdot i + 1)$ až $(d \cdot i + d)$. V našej implementácií navyše každý vrchol obsahuje smerník na svojho otca a smerník na pole synov. S takouto reprezentáciou v poli zľava úplný d-árny strom znamená, že v poli, v ktorom je uložený, nie sú „diery“.

Operácie. Operácia insert(x) vloží vrchol s kľúčom x na najbližšie voľné miesto, tak, aby sa neporušila úplnosť stromu a poslednej vrstvy. V praxi to znamená, že sa pridá nový prvok na koniec poľa. Takto vložený prvok môže porušovať podmienku haldy, takže ešte musí „prebublať“ smerom hore na správne miesto. Vymieňa sa so svojím otcom, až pokým nie je podmienka haldy splnená.

Minimum sa nachádza v koreni haldy. Operácia *deleteMin* najprv vymení koreň haldy s posledným vrcholom a potom minimum, ktoré sa teraz nachádza na konci haldy, odstráni. Koreň haldy po výmene nemusí spĺňať podmienku haldy a preto musí „prebublať“ nadol. Vymieňa sa so svojím najmenším synom, až pokým nie je splnená podmienka.

Po zavolaní operácie decreaseKey(v, Δ) vrchol v nemusí spĺňať podmienku haldy a preto musí opäť „prebublať“ nahor.

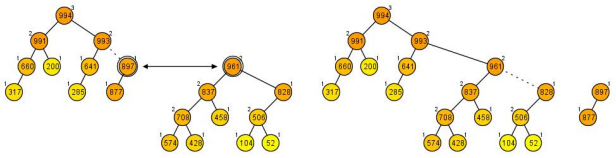
Časová zložitosť. Z popisu jednotlivých operácií sú zrejmé časové zložitosti. CreateHeap a findMin sa deje v konštantnom čase. Ak n je počet prvkov v halde, potom operácie insert a decreaseKey majú časovú zložitosť $O(log_d(n))$, pretože $log_d(n)$ je hĺbka stromu, teda vrchol by sa po $log_d(n)$ krokoch dostal ku koreňu. Operácia deleteMin má zložitosť $O(d \cdot log_d(n))$, pretože sa navyše pri prebublávaní musí hľadať najmenší syn spomedzi d synov.

Použitie. Z časových zložitostí platiacich pre binárnu haldu sa môže zdať vznik d-árnej haldy zbytočný. Avšak v mnohých reálnych prípadoch funguje zovšeobecnená verzia efektívnejšie. Jednak sú to príklady, keď operácie insert a decreaseKey sú využívané častejšie ako operácia deleteMin. Napríklad v Dijkstrovom algoritme pre najkratšie cesty v grafe funguje v určitých prípadoch d-árna halda rýchlejšie. Navyše, pre niektoré konkrétne d funguje d-árna halda rýchlejšie než binárna, vďaka tomu, že sa zníži počet prístupov na disk.

4.2 Ľavicová halda

Popis. Ľavicová halda je obmenou binárnej haldy. Navyše si pre každý vrchol pamätáme *rank*, čo je najkratšia vzdialenosť vrcholu k *externému vrcholu*. Každému vrcholu haldy, ktorému chýba aspoň jeden syn, sú doplnené špeciálne vrcholy tak, aby mal každý vrchol oboch synov. Týmto špeciálnym vrcholom hovoríme externé a nie sú súčasťou haldy. Ich rank je 0. Rank vrcholu x je daný rekurzívne ako $rank(x) = 1 + min\{rank(left(x)), rank(right(x))\}$. Pre ľavicovú haldu špeciálne platí, že rank pravého syna je menší alebo rovný ako rank ľavého syna. Toto zabezpečuje pre každý podstrom, že pravá cesta je vždy kratšia ako ľavá cesta.

Operácie. Najdôležitejšia operácia vykonávaná na Ľavicovej halde je *meld(i, j)*. Pomocou nej si zadefinujeme aj *insert(x)* a *deleteMin*. Haldy sa spájajú pozdĺž pravej cesty. Postupne prechádzame odvrchu nadol celú pravú cestu haldy i a porovnávame kľúče s koreňom haldy j. Ak narazíme na kľúč vrcholu v v halde i, ktorý je väčší ako kľúč v koreni w haldy j, vrcholy vymeníme. Teda z w sa stane pravý syn otca v a z v sa stane halda j. Kľúč prázdnej haldy považujeme za nekonečno. Takto pokračujeme, až kým nedôjdeme na koniec pravej cesty haldy i. Potom nasleduje fáza



Obr. 2: Spájanie pozdĺž pravej cesty

úpravy rankov. Ranky sa mohli zmeniť len na pravej, spájacej ceste, preto ich pozdĺž tejto cesty zdola nahor upravíme. Nakoniec pre ľavicovú haldu musí byť dodržané pravidlo o veľkosti rankov synov. Preto opäť prejdeme pravú cestu výslednej haldy a pokiaľ je niekde pravidlo porušené, bratov vymeníme.[2]

Pokiaľ máme definovanú operáciu *meld(i, j)*, zadefinovať *insert(x)* na halde i je jednoduché. Vytvorí sa nová jednoprvková halda j obsahujúca iba vrchol x a zavolá sa *meld(i, j)*.

Operácia *deleteMin* najprv vymaže vrchol haldy h a potom zavolá *meld(left(h), right(h))*.

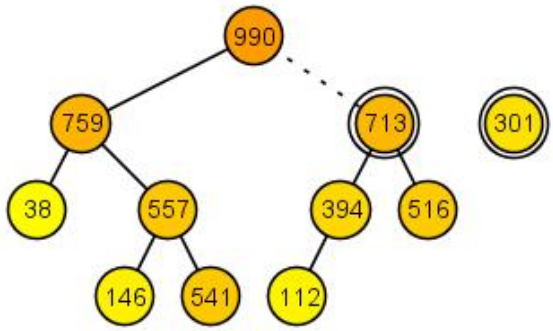
Hoci operácia *decreaseKey* nie je štandardná pre Ľavicovú haldu, v programe je definovaná ako *decreaseKey* pre binárnu haldu. Teda po znížení kľúča vrchol „prebubláva“ nahor.

Časová zložitosť. Veľkým plusom ľavicovej haldy je časová zložitosť pre spájanie $log(n)$, kde n je súčet vrcholov oboch háld, ktoré spájame. Toto sa dosiahne vďaka tomu, že cesta, pozdĺž ktorej sa dve haldy spájajú, sa udržuje čo najkratšia. *Insert(x)* a *deleteMin* sa majú rovnakú zložitosť ako *meld(i,j)*. *CreateHeap* a *findMin* majú konštantnú časovú zložitosť. Existuje lazy verzia Ľavicovej haldy (Tarjan (1983)), ktorá odkladá vymazávanie a spájanie. Časová zložitosť týchto dvoch operácií sa stane konšatntnou, na úkor operácie *findMin*. Tento druh sme však neimplementovali, preto sa ním nebudeme zaoberať.

4.3 Skew halda

Skew halda je jednou zo samoupravujúcich sa háld. To znamená, že negarantuje dobrú časovú zložitosť pre najhorší prípad, ale stará sa o to, aby v budúcnosti robila operácie efektívnejšie. Pri takomto druhu haldy sa pozeráme na *amortizovanú časovú zložitosť*. Nezaujíma nás časová zložitosť pre najhorší prípad, ale priemerná zložitosť postupnosti operácií.

[2] Druhý a tretí krok sa dajú robiť súčasne, avšak z hľadiska prehľadnosti vizualizácie sú v našom programe implementované po sebe.



Obr. 3: Spájanie pozdĺž pravej cesty

Popis. Skew halda je odvodená z Ľavicovej haldy. Jediný rozdiel je, že pre Skew haldu nedefinujeme rank. Teda je to opäť druh binárnej haldy. Taktiež jej hlavnou výhodou je spájanie, je však jednoduchšia na implementáciu.

Operácie. Prvá fáza operácie *meld(i,j)* na Skew halde je totožná s Ľavicovou haldou. Keďže ranky tu neexistujú, druhá fáza spájania Ľavicových háld sa preskočí a prejdeme k poslednej fáze. Táto časť obsahuje kľúčovú úpravu haldy, ktorá zabezpečuje efektívne spájanie. Postupujeme po pravej spájacej ceste haldy, ktorá vznikla v prvom kroku. Začneme v predposlednom vrchole[3] smerom nahor až po koreň. Každému vrcholu po ceste vymeníme synov.

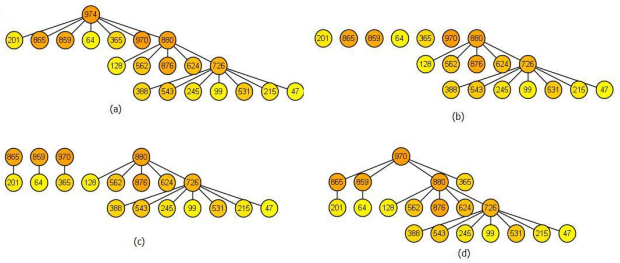
Zvyšné operácie sú definované rovnako, ako pri Ľavicovej halde.

Časová zložitosť. Amortizovaná časová zlotosť pre *meld(i,j)* je $O(log(n))$. Takisto platí aj pre *insert(x)*, *deleteMin* a *decreaseKey(v, Δ)*. *CreateHeap* a *findMin* majú konštantnú časovú zložitosť. Dôkazy časových zložitostí vyžadujú väčší priestor, sú popísané v Sleator and Tarjan (1986). Ku Skew halde tiež existujú alternatívy - Top-down, Bottom-up, pričom Bottom-up prístup ohraničuje všetky operácie až na *deleteMin* na $O(1)$. *DeleteMin* má v tomto prípade časovú zložitosť $O(log(n))$.

4.4 Párovacia halda

Popis. Párovacia halda je ďalším druhom samoupravujúcej sa haldy. Opäť sa budeme pozerať na amor-

[3]Keďže posledný vrchol nemá pravého syna, nemá zmysel mu vymieňať synov, nanajvýš by sme tým predĺžili pravú cestu.



Obr. 4: *Vymazanie minima* (a) pôvodná halda (b) halda po vymazaní minima (c) halda po párovaní (d) halda po spájaní

tizovanú časovú zložitosť jej operácií. Je to všeobecná halda, teda počet synov nie je obmedzený. Základná procedúra, ktorú táto dátová štruktúra implementuje je spájanie (*linking*) dvoch háld. Procedúra spočíva iba v tom, že sa halda s väčším kľúčom v koreni napojí na tú s menším kľúčom. V našej implementácií sa nový vrchol napája vždy ako prvý syn.

Operácie. Operácia *meld(i, j)* využíva procedúru pre linking. Tiež *insert(x)*, len prilinkuje novú jednoprvkovú haldu. Operácia *decreaseKey(v, Δ)* najprv zníži hodnotu vrcholu v, a keďže môže byť porušená podmienka pre haldu, strom zakorenený vo vrchole v sa odtrhne a prilinkuje ku zvyšku. Časové zložitosti pre všetky tieto operácie sú $O(1)$. Najzaujímavejšie na Párovacích haldách je *deleteMin*. Po odstránení koreňa ostane les jeho detí. Môžeme zvoliť niekoľko prístupov ako z detí vytvoríme nový strom.

Naivné riešenie hovorí, že si vyberieme jedno dieťa a ostatné k nemu prilinkujeme. Už na prvý pohľad vidíme, že pri nesprávnom zvolení prvého dieťaťa môže byť časová zložitosť takéhoto algoritmu $O(n)$.

Ďalší, o niečo lepší nápad je deti najprv popárovať a prilinkovať. S výhliadkami do budúcnosti nám tento algoritmus dá amortizovanú časovú zložitosť $O(\sqrt{n})$.

Keď si dáme väčší pozor na to, ako deti párujeme, môžeme dosiahnnuť lepšie výsledky. Pokiaľ párujeme synov v poradí v akom boli prilinkovaní od najmladšieho a potom ich sprava doľava prilinkujeme, vieme dostať lepšie, avšak dosiaľ nedokázané výsledky. Ostatné riešenia sme zatiaľ neimplementovali, preto len spomenieme, že sú popísané napríklad v Fredman et al. (1986).

Aby boli ale operácie efektívne, musíme zvoliť správnu reprezentáciu tejto dátovej štruktúry. V našom programe sme použili reprezentáciu pomocou binárneho stromu (*binary tree represetation*), ktorú na-

programoval Viktor Tomkovič. V každom vrchole sa uchováva ľavý smerník na prvého syna, pravý smerník na nasledujúceho brata a ešte jeden smerník na rodiča.

Vizualizácia. Aby sa dala vizualizácia *deleteMin* lepšie previesť, minimum, teda koreň haldy ostáva súčasťou haldy až do konca operácie, ale je zneviditeľnený. Navonok teda vyzerá, že po odstránení minima ostal les synov, ale v skutočnosti je to stále jedna halda. Týmto využijeme už naprogramované rozloženie vrcholov a nemusíme zavádzať nové pole pre synov.

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 $p(x) =$ `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.

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).

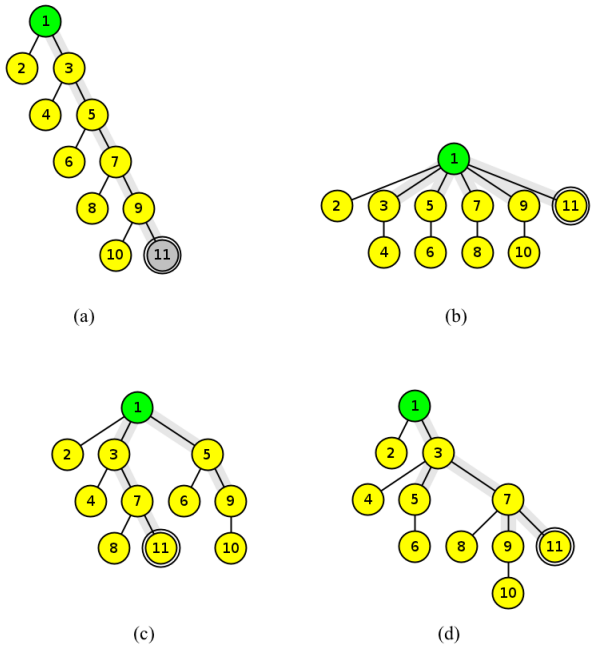
Existujú dva prístupy ako zlepšiť operácie a tým aj zrýchliť ich vykonanie. Sú to: heuristika *spájanie podľa ranku* a rôzne heuristiky na *kompresiu cesty*.

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[4] 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[5] v ceste od vrcholu x po koreň stromu na otca jeho otca.

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

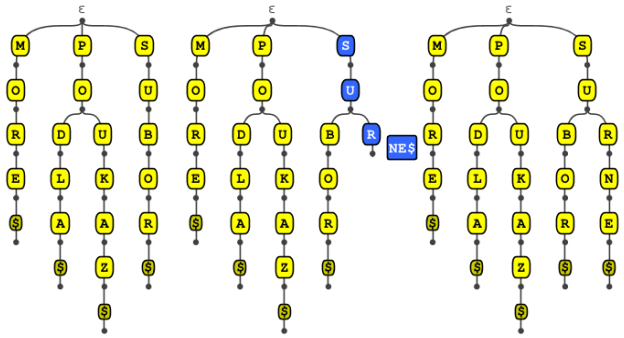
[5] Okrem koreňa a synov koreňa, keďže tie deda a otca resp. deda nemajú.



Obr. 5: *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.

Časová zložitosť union-findu záleží od toho, koľko prvkov je v množinách a koľko je operácií celkovo vykonaných operácií. Všetky uvedené spôsoby ako 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 (Walker II, 1990). 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.



Obr. 6: *Vloženie slova „*SURNE*“.* Začiatok slova „SU“ sa v strome nachádza, ešte treba pripojiť hrany so znakmi R, N, E a $.

6 Písmenkový strom

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

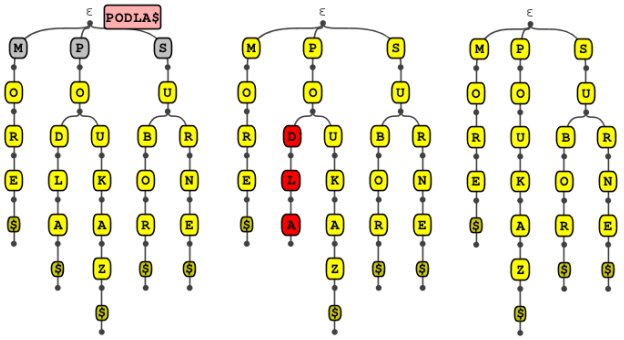
- $insert(w)$ – pridá do stromu slovo w;
- $find(w)$ – zistí, či sa v strome slovo w nachádza;
- $delete(w)$ – odstráni zo stromu slovo w.

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. 6).

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



Obr. 7: *Odstránenie slova „*PODLA*“*. Po odstránení $ nám v strome ostane nepotrebá prípona „DLA“ (mŕtva vetva), ktorá je vyznačená červenou.

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. 7).

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*[6], 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[7], 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. (Walker II, 1990) 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 História

Každá vizualizovaná operácia (insert/delete/...) 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 operácie 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 ľubo-

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

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

voľný 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.

8 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 „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.

8.1 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. W. and Jacobson, G. J. (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.

Fredman, M. L., Sedgewick, R., Sleator, D. D., and Tarjan, R. E. (1986). The pairing heap: A new form of self-adjusting heap. *Algorithmica*, pages 111–129.

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. M. (1983). *Word hy-phen-a-tion by comput-er*. 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. M. and Tilford, J. S. (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.

Sleator, D. D. and Tarjan, R. E. (1986). Self-adjusting heaps. *SIAM J. COMPUT.*

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

Tarjan, R. E. (1983). *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1st edition.

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.