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

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).

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`.j 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*.

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.

TODO: V sekcií 2 popiseme toto, v dalsich sekciach tamto

2 Rozšírenie predošlej práce

Projekt Gnarley Trees sme rozšírili nielen o vizualizácie ďalších dátových štruktúr, ale pribudli aj soft-

*`algvis@googlegroups.com`

vérové (vizualizačné) vylepšenia.

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 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.

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

Pôvodná verzia (Kováč, 2007) obsahovala vizualizáciu viacerých vyvážených stromov. K nim sme pridali B^+-stromy, stromy s prstom a stromy s reverzami.

3.1 B^+-strom

Popis. B^+-*strom* je variácia B-stromu, v ktorom sú všetky kľúče uložené v listoch a listy sú pospájané do spájaného zoznamu. B^+-strom rádu B je strom, v ktorom má každý vnútorný vrchol najmenej $\lfloor B/2 \rfloor$, ale najviac B synov. Vďaka tomu je dobre vyvážený a jeho operácie sú vykonávané v logaritmickom čase. B^+-strom je *asociatívne pole (slovník)*, čiže poskytuje tieto tri operácie:

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

Operácia $find(x)$ 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 vrcholu. Je zrejmé, že ak sa väčší kľúč nenájde, presunieme sa do posledného, B-tého syna. V liste sa už len skontroluje, či sa v ňom hľadaný kľúč nachádza.

Definujeme dve operácie: COPY-UP a PUSH-UP, ktoré používa operácia $insert(x)$. 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 $insert(x)$ najprv pomocou operácie $find(x)$ zistí, či štruktúra daný kľúč obsahuje. Ak nie, je zrejmé, že patrí práve do vrchola, kde $find(x)$ skončil. Ak má vrchol 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 vrcholu. Ak otcovský vrchol presiahol najväčšiu možnú veľkosť, znova sa aplikuje popísaný algoritmus s jedným rozdielom – namiesto COPY-UP sa použije PUSH-UP.

Operácia $delete(x)$ najprv pomocou $find(x)$ 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 vrcholu, 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.

TODO: toto je pocet pristupov na disk! pocet krokov je $O(B\log_B n)$; ale pocet pristupov na disk nas zaujima, lebo disk je pomaly

Vkladanie, vymazávanie a hľadanie má časovú zložitosť $O(\log_B n)$.

Použitie. Hlavné využitie B$^+$-stromov je v databázových systémoch. Ak zvolíme vhodný rád, vieme jednotlivé vrcholy dobre napasovať na stránky a tým regulovať ako počet prístupov k pamäti, tak jej zaplnenie D. Mehta (2005). Agregačné funkcie, ako napríklad súčet, minimum, priemer, vieme pre daný interval spočítať v čase $O(\log_B(n))$. Vypísať všetky prvky z daného intervalu dokážeme pomocou $O(\log_B(n)+t/B)$ prístupov na disk.

Ďalšia výhoda B$^+$-stromov oproti B-stromom sa prejaví, ak máme utriedený zoznam dát a chceme z neho vytvoriť B$^+$-strom: B$^+$-strom môžeme vystavať odspodu. Takýto postup nám zaručí vyžaduje $O((n/B)\cdot\log_B n)$ prístupov na disk, čo je B-krát rýchlejšie ako spraviť n volaní $insert(x)$.

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.

TODO: toto je facina; samotny finger dobru zlozitost nezaruci (ako sa tana mohla sama presvedcit, ked sa to snazila implementovať cez 2-3-4 strom... strom ma finger property (vyhladavanie s prstom?) AK vie vyhladavat v $O(\log d)$ (namiesto trivialneho $O(\log n)$.

Popis. Strom s prstom je upravený 2-3-4$^+$ strom (B$^+$-strom rádu 4, t.j. 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 spojíme všetky vrcholy na rovnakej úrovni (v rovnakej vzdialenosti od koreňa) 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.

Operácia $find(x)$ začne na mieste, kam ukazuje prst. Skontroluje, či by kľúč mal patriť do daného vrcholu. 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.

Operácia $insert(x)$ najprv pomocou operácie $find(x)$ 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 $delete(x)$ najprv pomocou operácie $find(x)$ 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 aspoň dvoch synov, 2-3-4 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 $O(1)$ D. Mehta (2005).

Vizualizácia. Strom s prstom je vizualizovaný pomocou B^+-stromu s rádom 4, keďže jeho podmienky pre počet potomkov vyhovujú danej štruktúre.

TODO: OBRAZOK

3.3 Strom s reverzami

Strom s reverzami je dátová štruktúra na uchovávanie permutácií. Poskytuje operácie

- $find(k)$ – zistí, ktorý prvok je na k-tom mieste permutácie a
- $reverse(i, j)$ – reverzne interval od i po j.

Popis. Permutáciu reprezentujeme ako strom, v ktorom je in-order poradie prvkov totožné s poradím prvkov v permutácií. Strom s reverzami môžeme implementovať pomocou ľubovoľného vyváženého stromu, ktorý podporuje rozdelenie a zreťazenie dvoch stromov v logaritmickom čase. My sme zvolili splay strom pre jeho jednoduchosť.

Aby sme vedeli efektívne vyhľadať k-ty prvok, budeme si pre každý vrchol udržiavať veľkosť jeho podstromu. V operácií $find(k)$ sa vieme podľa toho rozhodnúť, či sa k-ty prvok nachádza v ľavom podstrome, resp. koľký prvok je v pravom podstrome. Po nájdení sa prvok presunie do koreňa pomocou operácie splay.

Operáciu $reverse(i, j)$ implementujeme lenivo: strom najskôr rozdelíme na tri časti: T_1, T_2, T_3, pričom T_2 obsahuje interval od i-teho po j-ty prvok, T_1 obsahuje začiatok a T_3 koniec permutácie. Koreň T_2 jednoducho označíme vlajkou, ktorá bude signalizovať, že podstrom je reverznutý a prvky sú v skutočnosti v opačnom poradí ako doteraz. Ak už koreň vlajku obsahuje, odstránime ju. Následne stromy T_1, T_2, T_3 opäť spojíme.

Pri takomto riešení musíme ešte upraviť vyhľadávanie a rotácie, aby brali do úvahy vlajky vo vrcholoch. Najelegantnejšie riešenie je odstrániť vlajku vždy, keď na ňu narazíme: Danému vrcholu odstránime vlajku, vymeníme mu synov a každému synovi vlajkový bit znegujeme.

Všetky operácie vieme implementovať v rovnakom čase ako operácie v splay tree, t.j. amortizovane $O(\log n)$.

Použitie. Stromy s reverzami (pôvodne založené na AVL stromoch) navrhli Chrobak et al. (1990) na efektívnu implementáciu 2-opt heuristiky na riešenie problému obchodného cestujúceho. Pri 2-opt heuristike sa snažíme reverzovať úseky cesty, kým nenájdeme lokálne minimum.

V bioinformatike sa tieto stromy používajú na triedenie orientovaných permuácií pomocou reverzov (Kaplan and Verbin, 2005; Swenson et al., 2009).

Za povšimnutie stojí fakt, že táto dátová štruktúra podporuje výmenu ľubovoľných dvoch blokov v logaritmickom čase, keďže túto operáciu vieme odsimulovať pomocou štyroch reverzov.

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 $reverse(i, j)$ nezostane ani T_1 ani T_4 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

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:

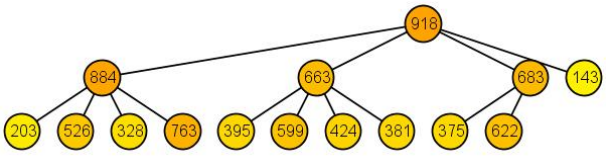
- **createHeap** – vytvorí prázdnu haldu;
- **insert** (x) – vloží vrchol s kľúčom x;
- **findMin** – vráti minimum t.j. hodnotu kľúča v koreni;
- **deleteMin** – odstráni vrchol s najmenším kľúčom, t.j. koreň;
- **decreaseKey** (v, Δ) – zníži kľúč vrcholu v o delta;

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.



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

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.

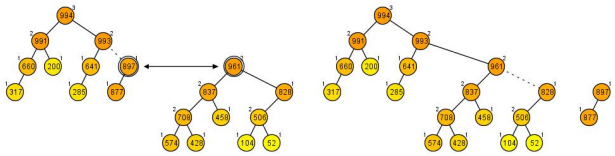
Časová zložitosť. Z popisu jednotlivých operácií sú zrejmé časové zložitosti. CreateHeap a findMin sa deje v konštantnom čase. Ak n je počet prvkov v halde, potom operácie insert a decreaseKey majú časovú zložitosť $O(log_d(n))$, pretože $log_d(n)$ je hĺbka stromu, teda vrchol by sa po $log_d(n)$ krokoch 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

[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. 2: Spájanie pozdĺž pravej cesty

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 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.

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.

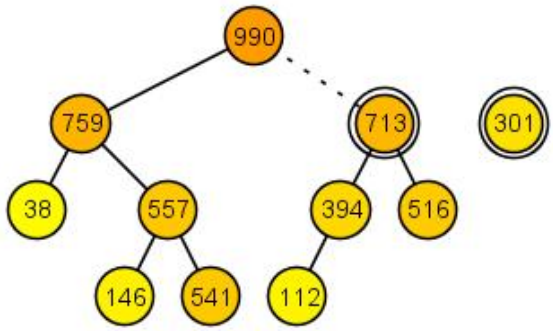
Operácie. Prvá fáza operácie *meld(i,j)* na Skew halde je totožná s Ľavicovou haldou. Keďže ranky tu neexistujú, druhá fáza spájania Ľavicových háld sa preskočí a prejdeme k poslednej fáze. Táto časť obsahuje kľúčovú úpravu haldy, ktorá zabezpečuje efektívne spájanie. Postupujeme po pravej spájacej ceste 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*

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

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



Obr. 3: Spájanie pozdĺž pravej cesty

na $O(1)$. *DeleteMin* má v tomto prípade časovú zložitosť $O(log(n))$.

4.4 Párovacia halda

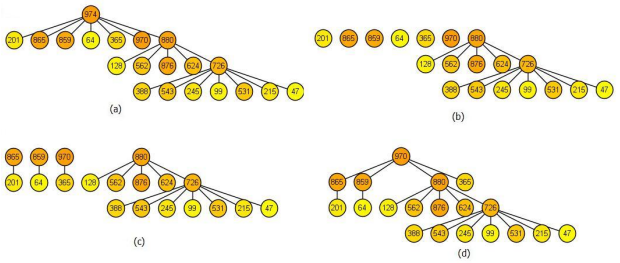
Popis. Párovacia halda je ďalším druhom samoupravujúcej sa haldy. Opäť sa budeme pozerať na amortizovanú časovú zložitosť jej operácií. Je to všeobecná halda, teda počet synov nie je obmedzený. Základná procedúra, ktorú táto dátová štruktúra implementuje je spájanie (*linking*) dvoch háld. Procedúra 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 dosiahnuť lepšie výsledky. Pokiaľ párujeme synov v poradí v akom boli prilinkovaní od najmladšieho a potom ich sprava doľava prilinkujeme, vieme



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

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 naprogramoval 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 $p(x) = \texttt{NULL}$).

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.

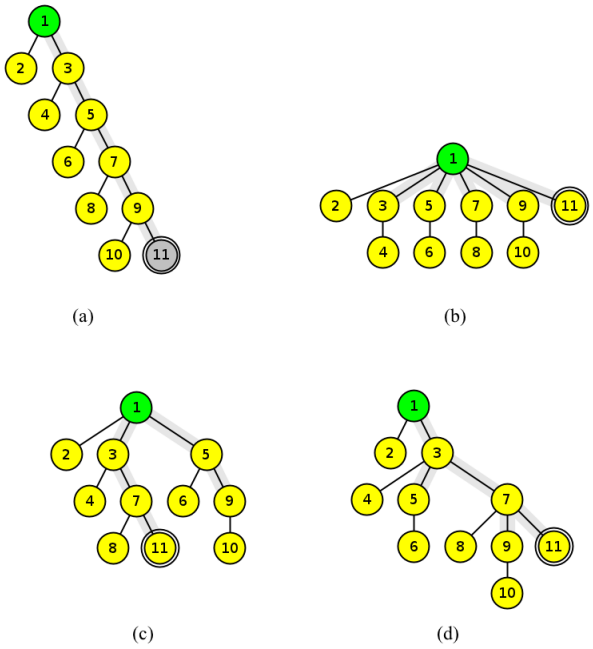
Použitie. Union-find sa dá použiť na reprezentáciu neorientovaného grafu, do ktorého pridávame hrany a odpovedáme na otázku „sú dané dva vrcholy spojené nejakou cestou?“ (t.j. sú v rovnakom komponente súvislosti?). Medzi najznámejšie aplikácie patria Kruskalov algoritmus na nájdenie najlacnejšej kostry (Kruskal, 1956) a unifikácia (Knight, 1989).

Gilbert et al. (1994) ukázali, ako sa dá union-find použiť pri Choleského dekompozícií riedkych matíc. Autori navrhli efektívny algoritmus, ktorý zistí počet nenulových prvkov v každom riadku a stĺpci výslednej matice, čo slúži na efektívnu alokáciu pamäte.

Pre offline verziu úlohy, kde sú všetky operácie dopredu známe, (Gabow and Tarjan, 1985) navrhli lineárny algoritmus. Článok obsahuje tiež viacero aplikácií v teoretickej informatike.

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.



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.

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, napojíme všetky vrcholy po ceste priamo pod koreň. Toto síce trochu spomalí prvé hľadanie, ale výrazne zrýchli ďalšie hľadania pre všetky prvky na ceste ku koreňu. 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 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 uživateľ vidieť, ako dátová štruktúra vyzerá, po vykonaní veľa operácií.

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).

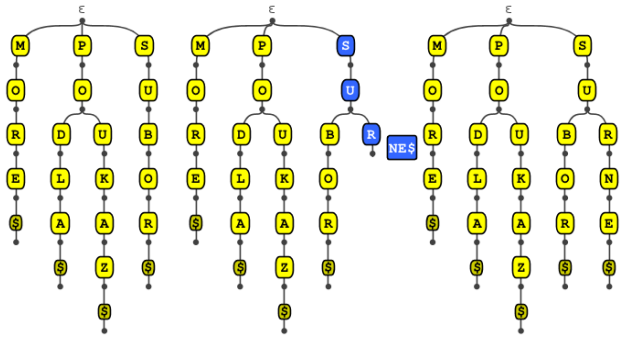
Popis. Písmenkový strom je *zakorenený strom*, v ktorom každá hrana obsahuje práve jeden znak z abecedy alebo *ukončovací znak*. Teda, každá cesta z koreňa do listu so znakmi $w_1, w_2, \ldots, w_n, \$$ prirodzene zodpovedá slovu $w = w_1 w_2 \cdots w_n$. *Ukončovací znak* je ľubovoľný, dopredu dohodnutý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

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

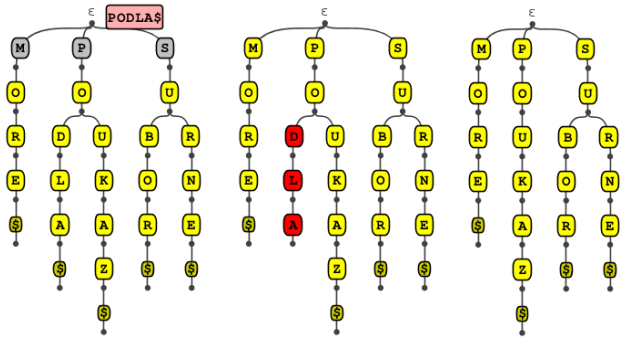
- $insert(w)$ – pridá do stromu slovo w;
- $find(w)$ – zistí, či sa v strome slovo w nachádza;
- $delete(w)$ – odstráni zo stromu slovo w.

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).



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



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

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. 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*[4], keďže

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

išlo o spôsob udržiavania dát v pamäti.

Morrison (1968) navrhol písmenkový strom, v ktorom sa každá cesta bez vetvení skomprimuje do jedinej hrany (na hranách potom nie sú znaky, ale slová). Táto štruktúra je známa pod menom *PATRICIA* (tiež *radix tree*, resp. *radix trie*) a využíva sa napríklad v *routovacích tabuľkách* (Sklower, 1991).

Písmenkový strom (tzv. *packed trie* alebo *hash trie*) sa používa napríklad v programe TEX na slabikovanie slov (Liang, 1983). Pôvodný návrh (Fredkin, 1960) ako uložiť trie do pamäte zaberal príliš veľa nevyužitého priestoru. Liang (1983) však navrhol, ako tieto nároky zmenšiť použitím hašovacej tabuľky.

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*.

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 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 dnárnej, 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.

Chrobak, M., Szymacha, T., and Krawczyk, A. (1990). A data structure useful for finding hamiltonian cycles. *Theoretical Computer Science*, 71(3):419–424.

D. Mehta, S. S. (2005). *Handbook of Data Structures and Applications*. Chapman and Hall/CRC Press, Boca Raton, New Yourk, Washington D.C., London.

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.

Gabow, H. and Tarjan, R. (1985). A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221.

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

Gilbert, J., Ng, E., and Peyton, B. (1994). An efficient algorithm to compute row and column counts for sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 15:1075.

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.

Kaplan, H. and Verbin, E. (2005). Sorting signed permutations by reversals, revisited. *Journal of Computer and System Sciences*, 70(3):321–341.

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, Univerzita Komenského v Bratislave.

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.*

Swenson, K., Rajan, V., Lin, Y., and Moret, B. (2009). Sorting signed permutations by inversions in $O(n \log n)$ time. In *Research in Computational Molecular Biology*, pages 386–399. Springer.

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.