Gnarley Trees: vizualizácia dátových štruktúr

Katka Kotrlová Pavol Lukča Viktor Tomkovič Tatiana Tóthová

Školiteľ: Jakub Kováč*

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

Abstrakt: V článku prezentujeme našu prácu na projekte Gnarley Trees, ktorý začal Jakub Kováč ako svoju bakalársku prácu. Zaoberá sa dátovými štruktúrami a ich vizualizáciou. Z vyvažovaných stromov to sú B^+-strom, strom s prstom a strom s reverzom, z háld d-árna a ľavicová halda, skew halda a párovacia halda. Z iných štruktúr bol pridaný union-find problém a písmenkový strom (trie). Okrem vizualizácie sme softvér doplnili o históriu krokov a operacií, jednoduchšie ovládanie a tesnejšie vykresľovanie vrcholov.

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 od implementačných detajlov a reprezentácie v pamäti. Je teda vhodnou pomôckou pri výučbe i samoštúdiu. Hoci výsledky výskumov zatiaľ nedokázali úplne preukázať pedagogickú efektívnosť vizualizácií (Shaffer et al., 2010), viacero štúdií potvrdilo zvýšený záujem a zapojenosť študentov (Naps et al., 2002; Hundhausen et al., 2002).

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).

Rozmach vizualizácie 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/`.

*`algvis@googlegroups.com`

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 *B^+-strom*, *strom s prstom* a *strom s reverzami*, z háld to sú *d-árna halda*, *ľavicová halda*, *skew halda* a *párovacia halda*. Taktiež vizualizujeme aj *union-find problém* a *písmenkový strom (trie)*.

Okrem vizualizácie softvér prerábame a neustále vylepšujeme. Doplnili sme ho o históriu krokov a operácií, jednoduchšie ovládanie a veľa ďalších funkcií. Softvér je celý v slovenčine aj angličtine a je implementovaný v jazyku `Java`. Dostupný je na stránke `http://people.ksp.sk/~kuko/gnarley-trees` vo forme appletov s jednotlivými dátovými štruktúrami, a tiež vo forme samostatného programu, ktorý obsahuje všetky dátové štruktúry a je určený na používanie offline.

Našou snahou je vytvoriť kvalitný softvér nezávislý od operačného systému, ktorý bude vyhovovať ako pomôcka pri výučbe ako aj pri samoštúdiu a bude voľne prístupný.

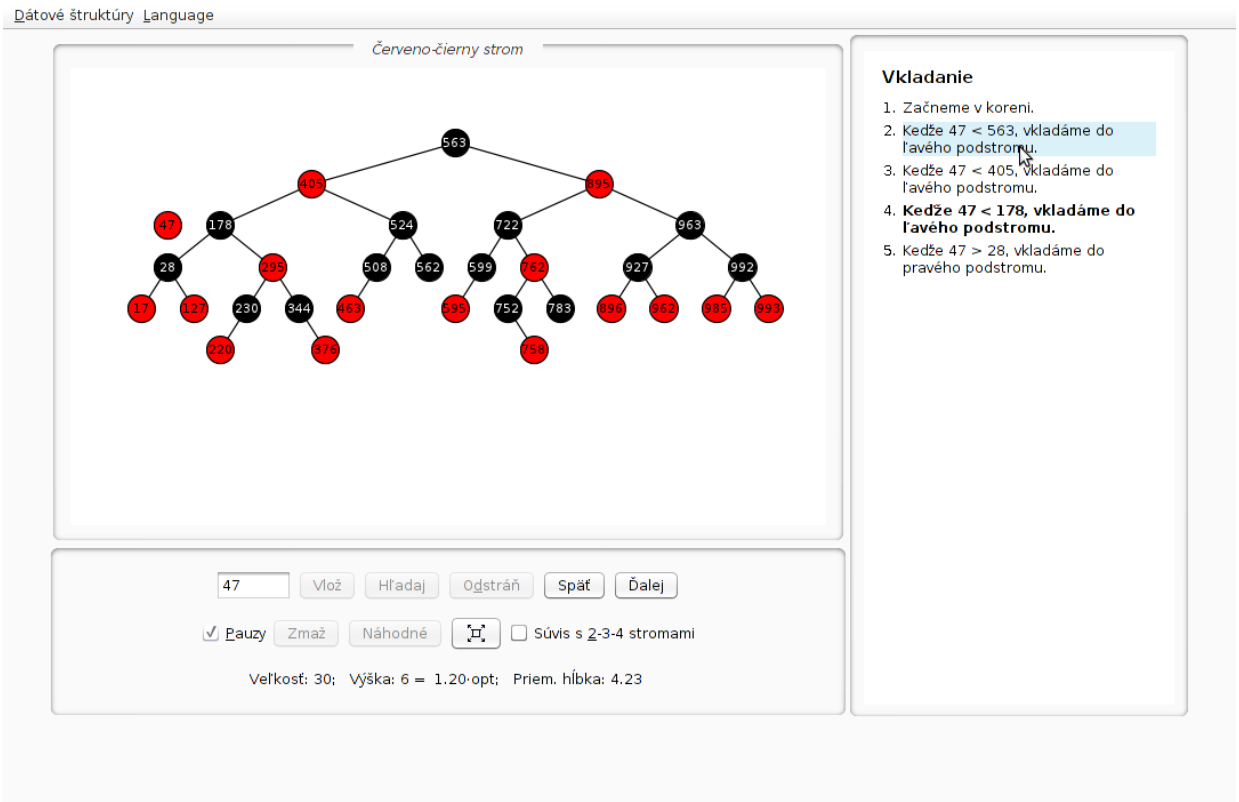
Zvyšok článku je organizovaný nasledovne: V sekcií 2 popisujeme implementované vylepšenia týkajúce sa vizualizácie, grafiky a ovládania, v sekciách 3 až 6 nové dátové štruktúry, ktoré sme implementovali a vizualizovali: vyvážené stromy, haldy, union-find a písmenkové stromy.

2 Rozšírenie predošlej práce

Projekt Gnarley Trees sme rozšírili nielen o vizualizácie ďalších dátových štruktúr, ale pribudli aj softvérové (vizualizačné) vylepšenia: kompaktnejšie vykresľovanie stromov, história krokov s možnosťou návratu, či približovanie a vzďaľovanie.

2.1 Tesnejšie vykresľovanie stromov

V pôvodnej verzii programu sa stromy vykresľovali tak, že vertikálna súradnica predstavovala hĺbku v



Obr. 1: *Softvér Gnarley Trees.* Vpravo je história krokov. V histórií sa dá navigovať buď pomocou tlačidiel „Späť“/„Ďalej“, alebo kliknutím na konkrétny krok v histórií.

strome a horizontálna poradie vrcholu v *inorderovom prechode* stromu. Tento jednoduchý spôsob však nešetrí priestor a pri štruktúrach ako union-find či písmenkový strom by výsledné stromy vyzerali škaredo. Preto sme sa rozhodli pre stromy implementovať alternatívny spôsob rozloženia, ktorý pre binárne stromy navrhli Reingold and Tilford (1981) a na všeobecné stromy rozšíril Walker II (1990). Tieto rozloženia vykresľujú vrcholy stromov čo najtesnejšie, pričom dodržujú tieto estetické 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 obrátime poradie všetkých synov, výsledný strom sa vykreslí zrkadlovo.

Reingoldov-Tilfordov, aj zovšeobecnený Walkerov algoritmus pracuje v lineárnom čase.

2.2 História krokov

Každá vizualizovaná operácia na dátovej štruktúre pozostáva z niekoľkých krokov. Jednou z noviniek v projekte je možnosť vrátiť sa pri prehliadaní operácií o niekoľko krokov späť, resp. vrátiť späť celé operácie.

Niekedy sa stáva, že nedočkavý používateľ 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. Používateľ má možnosť vrátiť sa späť o jeden krok (tlačidlo „Späť“/„Previous“) alebo preskočiť na ľubovoľný krok po kliknutí na zodpovedajúci komentár (pozri obr. 1).

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.

2.3 Ďalšie rozšírenia

K ďalším rozšíreniam patrí možnosť priblíženia, vzdialenia a presunu vykreslenej dátovej štruktúry v rámci vizualizačnej plochy. Používateľ 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. Používateľ si môže konkrétnu vizualizáciu pozrieť toľkokrát, 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. Prvky vo vnútorných vrcholoch slúžia len na navigáciu.

B^+-strom rádu b je strom, v ktorom má každý vnútorný vrchol najviac b a najmenej $\lfloor b/2 \rfloor$ synov (okrem koreňa, ktorý má najmenej dvoch 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 prvok x;
- $find(x)$ – zistí, či sa x v strome nachádza;
- $delete(x)$ – odstráni x zo stromu.

Operácia $find(x)$ začne v koreni, nájde v ňom prvý kľúč väčší od hľadaného. Nech je i-ty v poradí, potom hľadanie pokračuje i-tou vetvou. (Ak je x väčšie ako všetky kľúče, pokračujeme poslednou vetvou.) V liste už len skontrolujeme, či sa v ňom hľadaný kľúč nachádza.

Operácia $insert(x)$ najprv pomocou operácie *find* zistí, či štruktúra daný kľúč už obsahuje. Ak nie, je zrejmé, že x patrí práve na miesto, kde *find* skončil. Môže sa stať, že vrchol po vložení „pretečie“ – bude obsahovať $b+1$ prvkov. V takom prípade ak má vrchol brata s menej ako b prvkami, môžeme jeden kľúč presunúť k nemu. Ak sú susedné vrcholy plné, vrchol rozdelíme na dva, pričom stredný kľúč skopírujeme k otcovi. (Ten sa môže tiež preplniť, čím vznikne kaskáda rozdelení, ktorá skončí v najhoršom prípade v koreni.)

Podobne v operácií *delete* môže vrchol „podtiecť“. Ak má suseda s aspoň $\lfloor b/2 \rfloor + 1$ kľúčmi, môže si jeden požičať od neho. V opačnom prípade môžeme vrchol s jeho susedom zlúčiť.

Časová zložitosť a použitie. B^+-stromy sú vhodnou dátovou štruktúrou pre dáta uložené na disku: keďže dĺžka prístupu na disk je v porovnaní s výpočtami v hlavnej pamäti veľmi veľká, snažíme sa ich počet minimalizovať. Hoci časová zložitosť všetkých operácií je $O(b \log_b n)$, potrebujeme iba $O(\log_b n)$ prístupov na disk. Ak zvolíme vhodný rád, vieme jednotlivé vrcholy dobre napasovať na stránky a tým regulovať ako počet prístupov k pamäti, tak jej zaplnenie.

Hlavné využitie B^+-stromov je v databázových systémoch. Stromy vieme rozšíriť tak, aby podporovali rôzne agregačné funkcie, ako napríklad súčet, minimum, či priemer daného intervalu pomocou $O(\log_b n)$ prístupov na disk. 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 vyžaduje $O((n/b) \cdot \log_b n)$ prístupov na disk, čo je b-krát rýchlejšie ako spraviť n volaní *insert*.

3.2 Strom s prstom

Tradičné vyvážené vyhľadávacie stromy podporujú vyhľadávanie v čase $O(\log n)$. Prst je smerník na konkrétny vrchol, ktorý umožňuje efektívnejší prístup k vrcholom v jeho blízkom okolí. Hovoríme, že vyhľadávací strom podporuje vyhľadávanie s prstom (tzv. *finger search tree*), ak kľúč vo vzdialenosti d dokážeme nájsť v čase $O(\log d)$. Špeciálne predchodcu a následníka vieme nájsť v konštantnom čase.

Existuje viacero riešení, ktoré podporujú vyhľadávanie s prstom, v našom programe sme implementovali upravený 2-3-4$^+$ strom (Huddleston and Mehlhorn, 1982).

Popis. 2-3-4$^+$ strom je B$^+$-strom rádu 4, t.j. kľúče sú uložené v listoch a vnútorné vrcholy majú stupeň 2, 3 alebo 4. 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.

Operácia *find* začína v liste, na mieste, kam ukazuje prst. Vyhľadávanie pozostáva z dvoch fáz: V prvej fáze postupujeme nahor, až kým hľadaný kľúč nepatrí do nášho alebo susedovho podstromu. (V prípade potreby použijeme úrovňovú hranu, ktorou sa dostaneme do susedného vrcholu.) V druhej fáze potom zostupujeme nadol ako pri štandardnom vyhľadávaní. Pri *insert* a *delete* najskôr pomocou prstu nájdeme vhodné miesto a následne kľúč pridáme/vymažeme, rovnako ako v B$^+$-strome.

Takto implementované vyhľadávanie trvá $O(\log d)$, vkladanie a vymazávanie (po tom ako sme vrchol našli) má konštantnú amortizovanú zložitosť.

3.3 Strom s reverzami

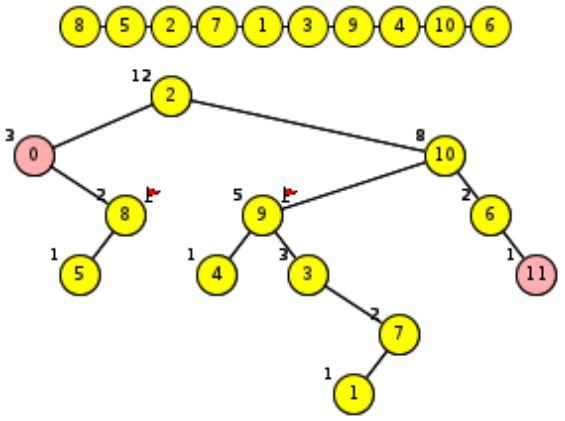
Strom s reverzami je dátová štruktúra na uchovávanie permutácií. Majme permutáciu π na množine $\{1,2,\ldots,n\}$; dátová štruktúra poskytuje operácie

- *reverse*(i,j) – preklopí poradie prvkov v intervale od i po j,
- *find*(k) – zistí, ktorý prvok je na k-tom mieste permutácie π.

Popis. Permutáciu reprezentujeme ako strom, v ktorom je *inorder* 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.

Niektoré vrcholy môžu byť označené vlajkou, ktorá signalizuje, že celý podstrom je reverznutý a prvky sú v skutočnosti v opačnom poradí (pozri obr. 2).

Operáciu *reverse* 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 (obr. 3). Koreň T_2 jednoducho označíme vlajkou. Ak už koreň vlajku obsahuje, odstránime ju. Následne stromy T_1, T_2, T_3 opäť spojíme.



Obr. 2: *Strom s reverzami.* Hore permutácia π, dolu reprezentácia pomocou splay stromu. Vlajky vo vrcholoch 8 a 9 signalizujú, že úseky $5,8$ a $4,9,3,1,7$ sú v opačnom poradí. Číslo vľavo hore od vrcholu je počet vrcholov v danom podstrome.

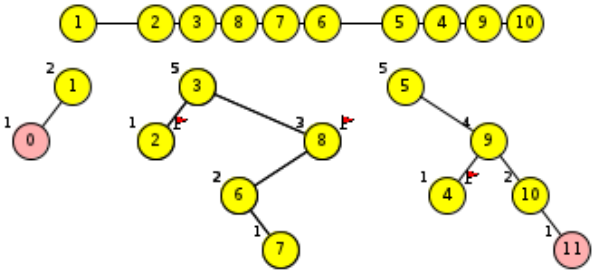
Aby sme vedeli efektívne vyhľadať k-ty prvok, budeme si pre každý vrchol udržiavať veľkosť jeho podstromu. V operácií *find*(k) sa vieme podľa toho rozhodnúť, či sa k-ty prvok nachádza v ľavom podstrome, resp. koľký prvok je v pravom podstrome. Po nájdení sa prvok presunie do koreňa pomocou operácie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

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, teda amortizovaná časová zložitosť oboch operácií je $O(\log n)$.

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 preklápať rôzne 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).



Obr. 3: Pri operácii *reverse* sa strom rozdelí na tri časti. Vľavo prvky pred intervalom, vpravo prvky za ním. Na reverznutie intervalu stačí pridať vlajku vrcholu 3 (koreň stredného stromu).

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 predstavu, bolo pridané pole, v ktorom užívateľ vidí skutočné poradie prvkov, ktoré zo stromu nie je až tak zjavné (obr. 2). Pole simuluje operácie spolu so stromom, tie sa však vykonávajú v lineárnom čase.

Do stromu sme pridali ako zarážky 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* nezostane T_1 ani T_3 prázdny.

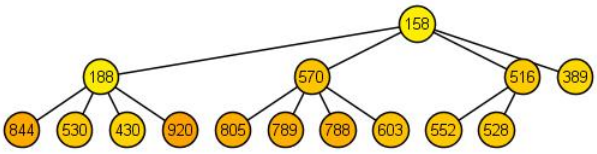
4 Haldy

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:

- *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. 4: *Príklad d-árnej haldy pre $d = 4$.* Vrcholy s menším kľúčom majú svetlejšiu farbu.

- *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 $\Delta \geq 0$;

Niektoré haldy navyše implementujú operáciu *meld*(i, j), ktorá 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 $(d \cdot i + d)$. 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.

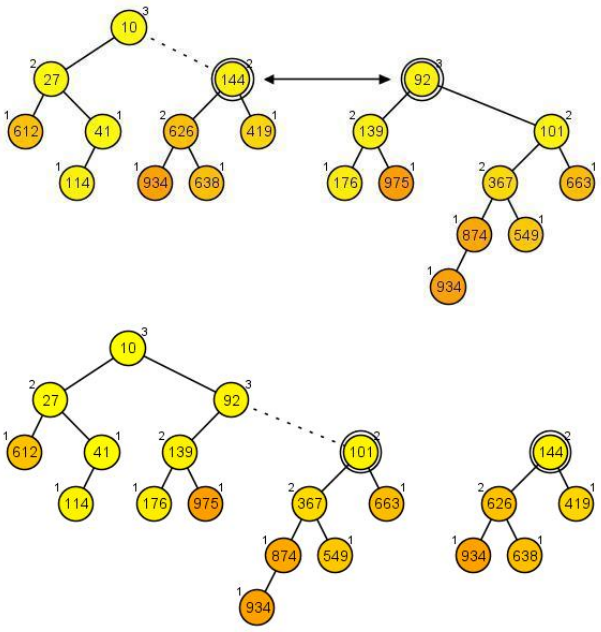
Minimum sa nachádza v koreni haldy. Operácia *deleteMin* najprv vymení koreň haldy s posledným vrcholom a potom minimum, ktoré sa teraz nachádza na konci haldy, odstráni. Koreň haldy po výmene nemusí spĺňať podmienku haldy a preto musí „prebublať“ nadol. Vymieňa sa so svojím najmenším synom, až pokým 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: 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 sa po $\log_d(n)$ krokoch dostane ku koreňu. Operácia *deleteMin* má zložitosť $O(d \cdot \log_d(n))$, pretože sa navyše pri prebublávaní musí hľadať najmenší syn spomedzi d synov; *findMin* sa deje v konštantnom čase.

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, najmä ak sú operácie *insert* a *decreaseKey* využívané častejšie ako operácia *deleteMin*. Ak napríklad v Dijkstrovom, či Jarníkovom–Primovom algoritme zvolíme $d = m/n$, vieme nájsť najkratšiu cestu, resp. minimálnu kostru v čase $O(m \log_{m/n} n)$.

Pre malé $d > 2$ funguje d-árna halda rýchlejšie než binárna vďaka lepšej práci s rýchlou vyrovnávacou pamäťou (cache); veľké d sú vhodné pre dátovú štruktúru uloženú na disku (kde sa počet prístupov na disk snažíme minimalizovať, podobne ako pri B-strome).



Obr. 5: *Spájanie pozdĺž pravej cesty.* V danom momente spájame podstrom 144 a 92 (hore). Aby sme zachovali podmienku haldy, pravý syn vrcholu 10 bude menší prvok z dvojice 144, 92. Preto tieto podstromy vymeníme a pokračujeme v spájaní podstromov 101 a 144 (dolu).

4.2 Ľavicová halda

Popis. V ľavicovej halde si pre každý vrchol pamätáme hodnotu *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 vrcholu w sa stane pravý syn otca v a z podstromu zakoreneným vrcholom 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.

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

Časová zložitosť. Veľkým plusom ľavicovej haldy je spájanie v logaritmickom čase. Toto sa dosiahne vďaka tomu, že cesta, pozdĺž ktorej sa dve haldy spájajú, sa udržuje čo najkratšia. Operácie $insert(x)$ a *deleteMin* majú rovnakú zložitosť ako $meld(i, j)$.

Existuje „lenivá“ verzia ľavicovej haldy (Tarjan, 1983), ktorá odkladá vymazávanie a spájanie na neskôr. Časová zložitosť týchto dvoch operácií sa stane konštantnou, na úkor operácie *findMin* (zložitosť je stále $O(\log n)$ ale iba v amortizovanom zmysle). 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.

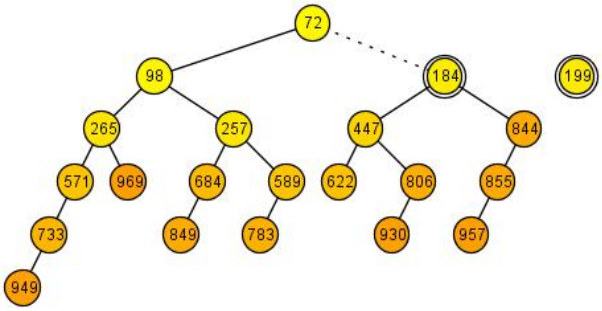
Popis. Skew halda je odvodená z ľavicovej haldy. Jediný rozdiel je, že pre skew haldu nedefinujeme rank. Teda je to opäť druh binárnej haldy. Taktiež jej hlavnou výhodou je spájanie, je však jednoduchšia na implementáciu.

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 $decreaseKey(v, \Delta)$ (Sleator and Tarjan, 1986). Ku skew halde tiež existujú alternatívy – top-down, bottom-up. Bottom-up prístup ohraničuje všetky operácie až na *deleteMin* na $O(1)$. *DeleteMin* má v tomto prípade časovú zložitosť $O(\log n)$.

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



Obr. 6: *Spájanie pozdĺž pravej cesty.*

4.4 Párovacia halda

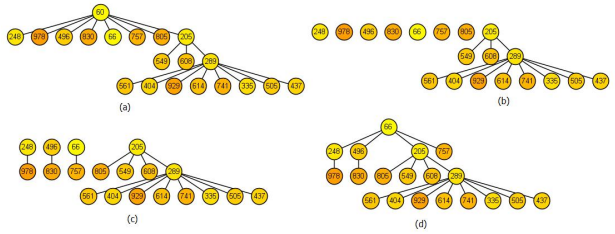
Popis. Párovacia halda je ďalším druhom samoupravujúcej sa haldy. Opäť sa budeme pozerať na amortizovanú časovú zložitosť jej operácií. Je to všeobecná halda, teda počet synov nie je obmedzený. Základná procedúra, ktorú táto dátová štruktúra implementuje je spájanie (*linking*) dvoch háld. Procedúra 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 $decreaseKey(v, \Delta)$ 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 dostať lepšie, avšak dosiaľ nedokázané výsledky. Iné riešenia sme zatiaľ neimplementovali. Niektoré z nich popísali Fredman et al. (1986).



Obr. 7: *Vymazanie minima.* (a) pôvodná halda, (b) halda po vymazaní minima, (c) halda po párovaní, (d) halda po spájaní.

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).

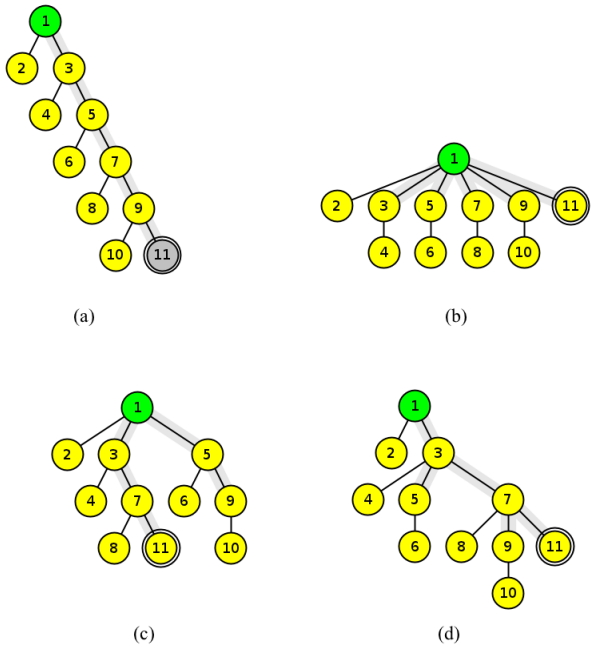
Gilbert et al. (1994) ukázali, ako sa dá union-find použiť pri Choleského dekompozícií riedkych matíc. Autori navrhli efektívny algoritmus, ktorý zistí počet nenulových prvkov v každom riadku a stĺpci výslednej matice, čo slúži na efektívnu alokáciu pamäte.

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. 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.



Obr. 8: *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.

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. Kompresie sú znázornené v obrázku 8.

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.

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:

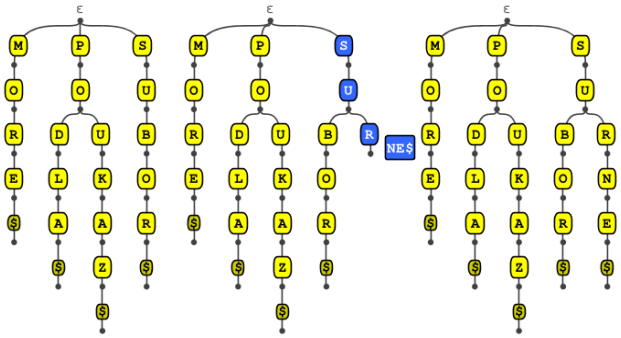
- $insert(w)$ – pridá do stromu slovo w;
- $find(w)$ – zistí, či sa v strome slovo w nachádza;
- $delete(w)$ – odstráni zo stromu slovo w.

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

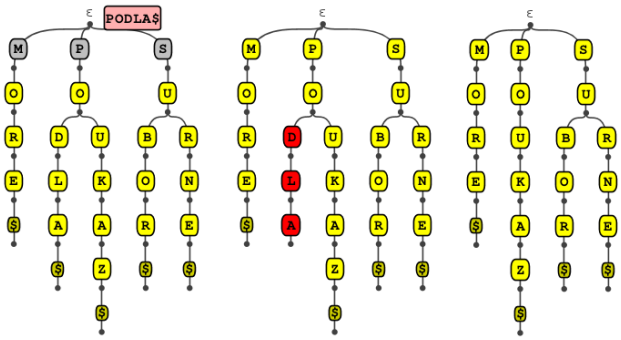
Operácia $insert(w)$ vloží do stromu vstupný reťazec tak, že z reťazca číta znaky a prechádza po príslušných hranách. Ak hrana s daným symbolom neexistuje, pridá ju (pozri obr. 9).

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



Obr. 9: *Vloženie slova „*SURNE*“*. Začiatok slova „SU“ sa v strome nachádza, ešte treba pripojiť hrany so znakmi R, N, E a $.



Obr. 10: *Odstránenie slova „*PODLA*“*. Po odstránení $ nám v strome ostane nepotrebá prípona „DLA“ (mŕtva vetva), ktorá je vyznačená červenou.

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. 10).

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 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.

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 doplňanie 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

V súčasnosti je málo programov, ktoré by prinášali komplexnejší prehľad využívaných dátových štruktúr. Na niektoré dokonca vizualizácia doposiaľ neexistuje. Je veľa appletov, ktoré implementujú niektoré algoritmy, avšak ich nedostatkom býva neprehľadnosť vizualizácie a hlavne okrem základných operácií dátovej štruktúry takmer žiadna interaktivita.

Do budúcna sa plánujeme zaoberať vizualizáciou ďalších dátových štruktúr a program obohatíme aj o známe algoritmy. Plánujeme implementovať *linking-cutting stromy*, *intervalové stromy*, *soft haldu* a niektoré *perzistentné dátové štruktúry*.

Budeme pokračovať v doplňaní histórie krokov do všetkých dátových štruktúr, ale aj vo vylepšení použí-

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

vateľského rozhrania, refaktorovaní zdrojového kódu a inými softvérovými vylepšeniami. Našim cieľom je čo najviac zjednodušiť prácu s programom, spraviť ho čo najviac užívateľsky prístupným a zrozumiteľným, 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, písmenkového stromu a implementoval algoritmus na vykreslovanie všeobecných stromov, Tatiana Tóthová vizualizovala B^+-strom, strom s prstom a strom s reverzami 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

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.

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.

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.

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

Huddleston, S. and Mehlhorn, K. (1982). A new data structure for representing sorted lists. *Acta informatica*, 17(2):157–184.

Hundhausen, C., Douglas, S., and Stasko, J. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290.

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.

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.

Naps, T., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., et al. (2002). Exploring

the role of visualization and engagement in computer science education. In *ACM SIGCSE Bulletin*, volume 35, pages 131–152. ACM.

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. (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.