DÁTOVÉ ŠTRUKTÚRY A ALGORITMY

Dokumentácia k druhému zadaniu

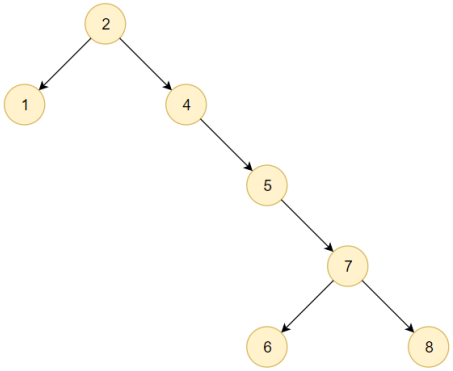
Vyhľadávanie v dynamických množinách

Autor: Adam Šípka

Popis použitých dátových štruktúr a ich implementácie

Nevyvážený binárny vyhľadávací strom - vkladanie

Prvý prvok, ktorý bol do stromu vložený sa nazýva koreň a svoju pozíciu už nemení. Ostatné vložené prvky idú buď do vetvy naľavo od koreňa, ak sú menšie ako on, alebo napravo, ak sú väčšie. Takýmto spôsobom sa strom rozvetvuje a pri vyhľadávaní bude prístup k prvkom rýchlejší. To však nebude platiť vždy, pri nevyváženom strome záleží od toho, v akom poradí sú prvky vložené. Ak by bol koreň príliš malý, alebo naopak príliš veľký, tak by na jednej strane stromu mohla vyniknúť dlhá vetva, vďaka ktorej, sa strácajú výhody binárnych vyhľadávacích stromov a daný strom bude skôr pripomínať spájaný zoznam. V najhoršom prípade, ak do stromu vkladáme postupnosť prvkov, tak nám vznikne spájaný zoznam. Tento problém riešia samovyvažovacie vyhľadávacie stromy. Vkladanie je v programe realizované pomocou rekurzie, ak je vkladaný prvok menší ako aktuálny, tak sa presunieme do ľavej vetvy, ak je väčší, tak do pravej. Keď bude aktuálny prvok NULL, tak vieme, že sme sa dostali na miesto, kde má byť nový prvok uložený a vkladanie môže úspešne skončiť.

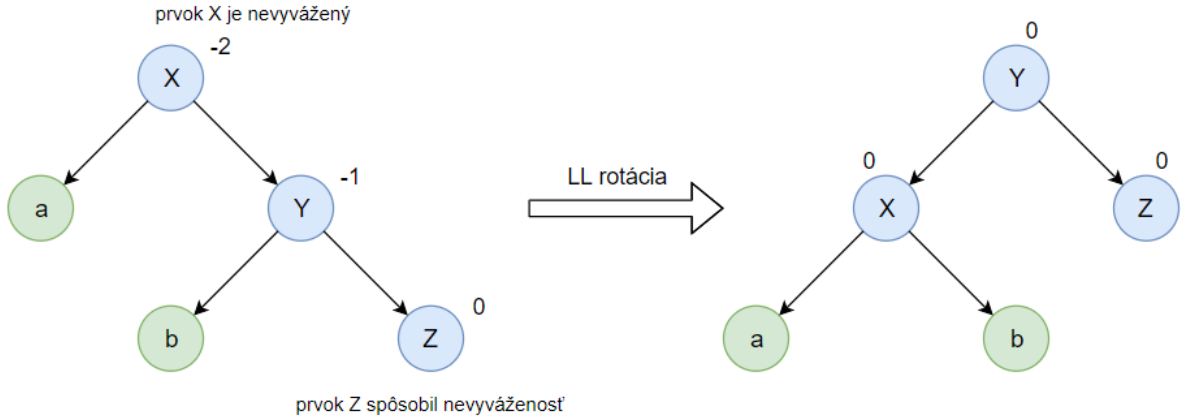


Vyhľadávanie v binárnych stromoch

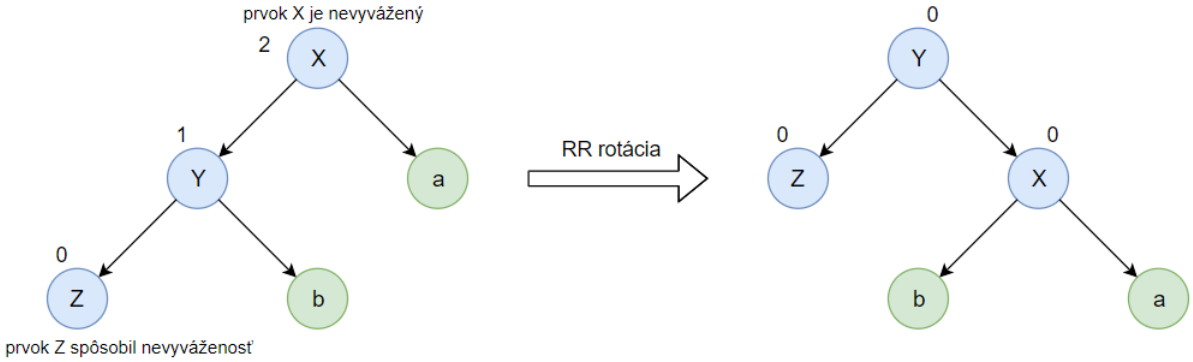
Vyhľadávanie prvkov je vo všetkých implementovaných binárnych stromoch rovnaké, jediné rozdiely sú pri vkladaní. Strom začneme prehľadávať od koreňa a aktuálny prvok vždy porovnáme s hľadaným. Podľa výsledkov porovnávania sa presunieme buď do pravej, ak hľadaný prvok je väčší, alebo ľavej vetvy, ak je menší, tak ako pri vkladaní. Ak dorazíme na prvok, ktorý má rovnakú hodnotu, ako ten, čo vyhľadávame, tak môžeme vyhľadávanie považovať za úspešne ukončené. Ak dorazíme na NULL, tak hľadaný prvok sa v strome nenachádza a funkciu môžeme tiež ukončiť. Vyhľadávanie je tiež, rovnako, ako vkladanie, realizované cez rekurziu, keďže v tomto prípade je to ľahšie na implementáciu a pochopenie, ako iné spôsoby.

AVL strom - vkladanie

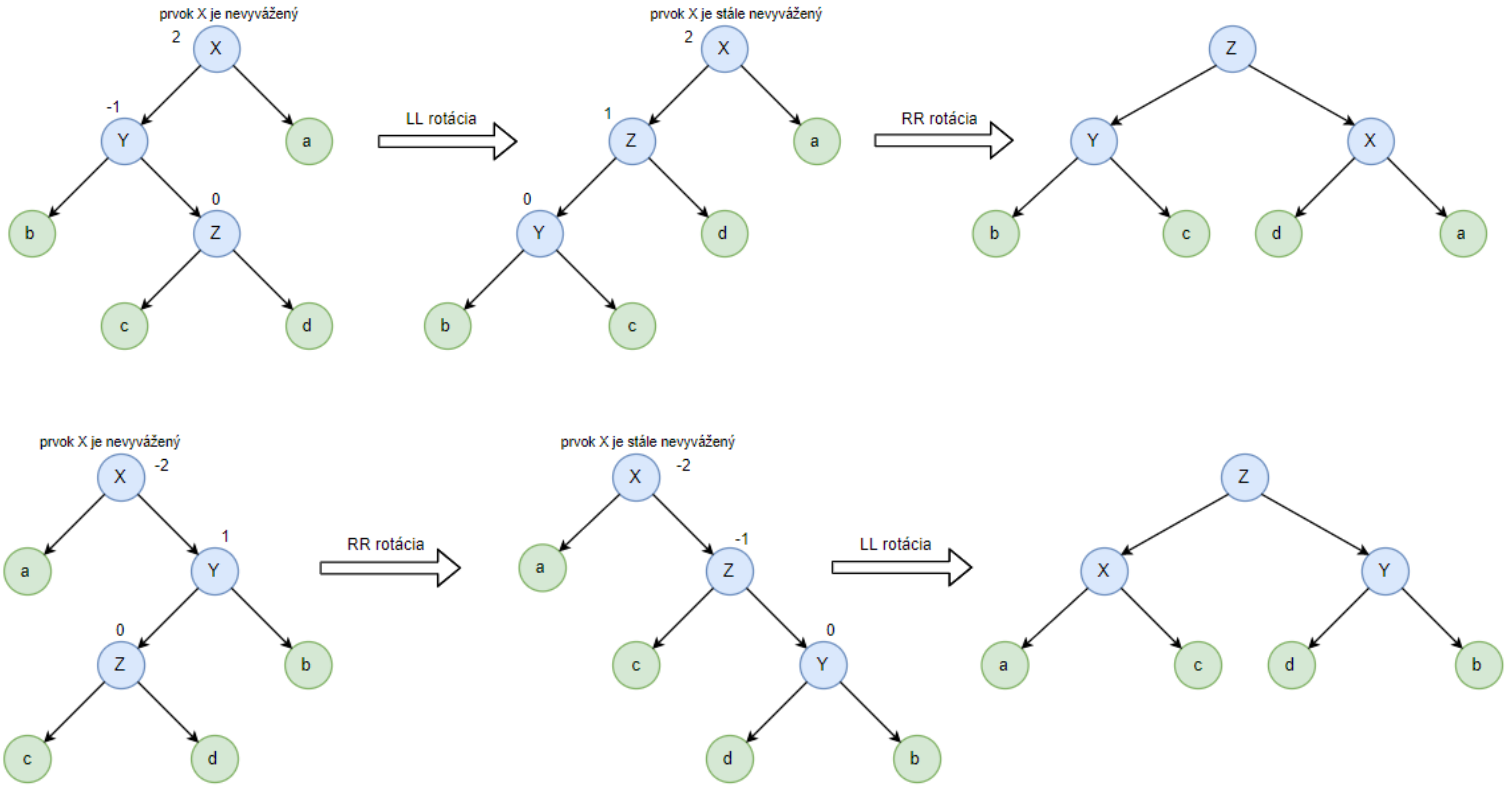
AVL strom je samovyvažovací binárny strom. Samotné vkladanie je rovnaké ako pri binárnych nevyvážených stromoch, ale po ňom môže dôjsť k rotácii niektorých problémových prvkov. Vďaka týmto rotáciam nám nikdy nevznikne spájaný zoznam. V AVL strome platia prísne pravidlá, ktoré sú po každom vložení kontrolované a ak sú porušené dochádza k rotácii, aby boli pravidlá zachované. Po vložení, keď sa postupne "vynárame" z rekurzie, tak vyrátame ***balance factor***, pre každý prvok, cez ktorý prechádzame. Ten sa ráta takým spôsobom, že si zistíme výšku pravého a ľavého podstromu a od ľavej výšky odčítame pravú. Ak je ***balance factor*** v intervale od -1 po 1, tak je všetko v poriadku a strom nemusíme vyvažovať. Pokiaľ má hodnotu -2, alebo 2, tak musíme niektoré prvky rotovať. Inú hodnotu nemôže mať, keďže hneď ako bude hodnota -2, alebo 2, dôjde k rotácii. V AVL strome existujú štyri typy rotácie, ktoré sú zobrazené nižšie.



Pri LL rotácii, ktorá vzniká, keď nevyváženosť spôsobil prvok v pravom podstrome, dochádza k presunutiu nevyváženého prvku doľava. Jeho pravé dieťa nahradí jeho pôvodné miesto a ľavý strom pravého dieťaťa sa stane pravým stromom kedysi nevyváženého prvku. Týmto nevyváženosť zaniká. Ďalší typ rotácie je RR rotácia, ktorá funguje takým istým spôsobom, len je opačne orientovaná.



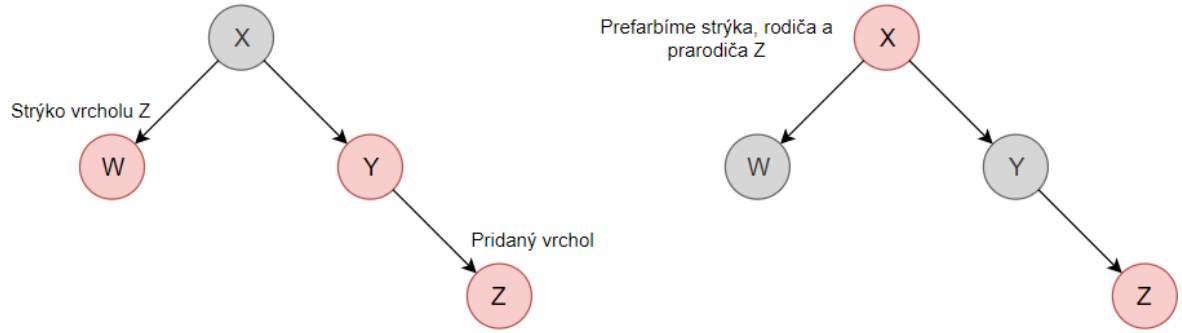
Ďalšie dva typy rotácii sú LR (left-right) a RL (right-left) rotácie. Pri LR rotácii najprv vykonáme LL rotáciu pre prvky Y a Z. Teraz je strom stále nevyvážený, ale keď vykonáme RR rotáciu pre Z a X, tak jeho vyváženosť bude obnovená. RL rotácia je to isté, len znova v opačnom poradí. Vďaka týmto rotáciám bude strom stále vyvážený a nevznikne nám lineárny zoznam, alebo dačo podobné, ako sa môže stať pri nevyvážených stromoch, takže pri vyhľadávaní sa k prvkom dostaneme skôr. Práve v tomto spočíva výhoda samovyvažovacích stromov.

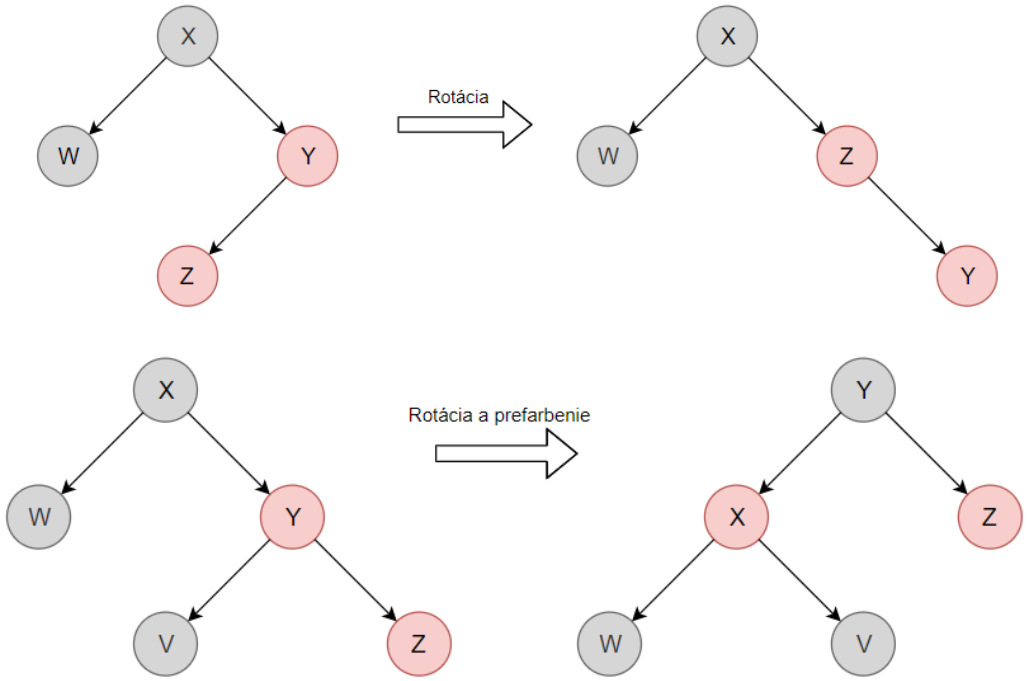


Červeno-čierny strom - vkladanie

Červeno-čierny strom, rovnako ako AVL strom, je samovyvažovací, avšak sú v ňom uplatnené iné pravidlá, ako v AVL strome. Každý prvok stromu má svoju farbu, buď červenú, alebo čiernu. Koreň stromu musí byť vždy čierny a listy taktiež, každý červený prvok musí mať len čierne deti, a každá cesta z ľubovoľného vrcholu do jedného z jeho podvrcholov musí obsahovať rovnaký počet čiernych prvkov. Tiež obsahuje rotácie ako AVL strom, ale podmienky na začatie rotácii sú iné, keďže tu normálna výška stromu a jeho balance factor nehrajú žiadnu rolu, lebo jeho vyváženosť je zabezpečená práve farbami jednotlivých vrcholov, ktoré zaisťujú, že akákoľvek cesta od koreňa po list nie je viac ako dvakrát tak dlhá ako akákoľvek iná cesta.

Vkladanie do stromu prebieha podobne, ako pri iných binárnych stromoch. Pridávaný vrchol uložíme na správne miesto rovnakým spôsobom, ako pri ostatných typoch stromov. Keď bude nový vrchol pridaný, tak mu farbu nastavíme na červenú a vytvoríme mu naľavo aj napravo čierne listy. Potom kontrolujeme, či sú splnené pravidlá, a ak nie sú, dochádza k navráteniu vyváženia stromu, buď pomocou prefarbenia, rotácie, alebo prefarbenia a aj rotácie. Znova sú tu rovnaké rotácie, ako pri AVL stromoch, ale dochádza k nim za iných podmienok. Môže nastať aj taká situácia, keď nemusí dôjsť k žiadnej rotácii a prvky v strome stačí len prefarbiť. Tá nastane vtedy, keď je strýko pridaného prvku červený. V tejto situácii prefarbíme strýka, rodiča a prarodiča pridaného prvku.





Hashovanie so zreťazením – vkladanie

Pred vkladaním prvkov do hashovacej tabuľky najprv zadáme počet prvkov, ktoré chceme pridať a podľa toho sa nastaví veľkosť tabuľky. Nastaví sa na prvočíslo, ktoré je najbližšie k dvojnásobku pridávaných prvkov. Potom sa môže začať samotné pridávanie. Hneď na začiatku nám hashovacia funkcia vráti index v tabuľke, kde sa majú dáta uložiť. Funkcia je veľmi jednoduchá, delí kľúč, ktorý dostala ako argument, veľkosťou tabuľky a vráti zvyšok po delení, ktorý symbolizuje index. Potom na vrátený index uloží údaje. Avšak niekedy môže dôjsť ku kolízii, dochádza k nej vtedy, keď je na vrátenom mieste už niečo uložené. Tieto kolízie som riešil zreťazením, čiže na každom indexe sa môže vytvoriť spájaný zoznam, a na jeho koniec sa môže nový údaj uložiť. Týmto je vkladanie prvkov úspešne ukončené. Aby sa tabuľka nepreplnila a spájané zoznamy neboli príliš dlhé, tak pri určitých situáciách dochádza k vytvoreniu novej, väčšej tabuľky a presunu starých údajov do nej. Táto situácia nastáva vtedy, keď faktor naplnenia nadobudne hodnotu pod jednu polovicu. Novovytvorená tabuľka je približne dvakrát väčšia ako pôvodná. Keďže veľkosť tabuľky je už nová, tak aj hashovacia funkcia nám bude vracať nové výsledky pre staré prvky, ktoré presunieme do novej tabuľky. Vďaka tomu bude hustota obsadenosti redšia a prístup k prvkom rýchlejší.

Hashovanie so zreťazením – vyhľadávanie

Vyhľadávanie údajov je časovo nenáročné, keďže tabuľka je pole, a vyhľadanie jedného prvku v poli má zložitosť O(1). Vyhľadávame podľa kľúča, ktorý najprv vstúpi do hashovacej funkcie, aby sme dostali index poľa. Ak je na danom indexe NULL, tak sa hľadaný prvok v tabuľke nenachádza. Ak je na danom indexe niečo uložené a hodnota toho údaju sa rovná hodnote kľúča, tak bolo vyhľadanie úspešné. Ak na danom indexe niečo je, ale hodnoty sa nerovnajú, presunieme sa na ďalší prvok spájaného zoznamu, ktorý sa začína na danom indexe. Postupne prechádzame cez spájaný zoznam a hľadáme zhodu. Ak ju nájdeme, funkcia končí úspešne, ak nie, hľadaný údaj v tabuľke nie je.

Hashovanie s otvoreným adresovaním – vkladanie

V prevzatom hashovaní funguje pridávanie prvkov podobne, hashovacia funkcia je rovnaká, ale veľkosť tabuľky je statická, takže počet prvkov, ktoré sa do nej zmestia je obmedzený. Keby boli kolízie riešené spájaným zoznamom, tak by k takémuto problému nedošlo, no tabuľka by bola po pridaní veľkého množstva preplnená a čoraz menej efektívna. Ale v tomto prípade sú kolízie riešené otvoreným adresovaním. Ak je index, ktorý nám hashovacia funkcia vráti, už obsadený, tak sa posunieme na nasledujúci. Opakujeme to, pokiaľ nenájdeme index, kde ešte nebolo nič uložené. Riešenie kolízii takýmto spôsobom je jednoduché na implementáciu, ale v tomto konkrétnom programe nedochádza k zväčšovaniu tabuľky, takže sa môže stať taká situácia, pri ktorej nám dôjde miesto v tabuľke a už nebudeme schopní vložiť ďalšie prvky.

Hashovanie s otvoreným adresovaním – vyhľadávanie

Vyhľadávanie prvkov je podobné vkladaniu, kľúč, podľa ktorého prvok hľadáme, nám hashovacia funkcia spracuje na index v poli. Presunieme sa naň a porovnávame hodnotu nášho kľúča s hodnotou uloženou na danom indexe. Ak sa hodnoty rovnajú, prvok bol nájdený, ak nie, tak sa presunieme na ďalší index a porovnávame znova. Tento postup opakujeme dovtedy, pokiaľ hľadaný prvok nenájdeme, alebo sa nedostaneme na koniec tabuľky. V tom prípade sa funkcia končí neúspešne a vráti nám NULL.

Testovanie dátových štruktúr

Vo všetkých dátových štruktúrach som testoval rýchlosť pridávania a vyhľadávania prvkov a celkový počet vykonaných operácii pri týchto dvoch operáciách. Testoval som na rôzne veľkých vstupoch s rôznymi hodnotami. Z viacerých dosiahnutých výsledkov som spravil aritmetický priemer a tieto údaje som si zaznamenal do tabuliek, kde som zaznamenané údaje porovnával a vyvodil z nich, kedy je vhodné akú dátovú štruktúru použiť. Pri testovaní som použil dva typy vstupov, aritmetickú postupnosť a náhodné čísla v rôznych intervaloch, ktoré sa líšili od množstva pridávaných prvkov. Tieto vstupy som testoval na rôznom množstve pridávaných a vyhľadávaných prvkov (od 10,000 do 100,000), aby boli moje výsledky presnejšie.

Rýchlosť vkladania prvkov do dátových štruktúr

Ako prvú som testoval rýchlosť vkladania prvkov. Nevyvážený binárny strom vkladal náhodné prvky rýchlo, keďže tam nie je potreba po každom pridanom prvku kontrolovať stav celého stromu a vyvažovať ho, ako napríklad pri AVL strome, avšak pri horšom vstupe by sa rýchlosť pridávania mohla znížiť. Avšak, pri vkladaní postupnosti bol čas trvania príliš dlhý a pri 25,000 prvkoch presiahol 2 sekundy, takže na väčších vstupoch som ho už netestoval, keďže vlastne nešlo už o binárny vyhľadávací strom, ale o spájaný zoznam. Moja implementácia AVL stromu bola pri pridávaní náhodných čísel najpomalšia zo všetkých, keďže po každom pridanom prvku bolo treba skontrolovať celý strom a zistiť, či neporušuje pravidlá vyváženosti a ak áno, tak bolo potrebné vykonať rotácie, čo spomalilo celkovú rýchlosť pridávania. Keďže je strom samovyvažujúci, tak pridávanie číselnej postupnosti bolo podstatne rýchlejšie, ako pri nevyváženom strome, ale bolo mierne pomalšie, ako pridávanie náhodných prvkov. Červeno-čierny strom bol zo všetkých binárnych stromov najrýchlejší, lebo v ňom neplatia také prísne pravidlá, ako v AVL strome, takže by bolo najvýhodnejšie použiť ten, pokiaľ nám ide o rýchlosť vkladania a chceme použiť binárny strom. Avšak moje hashovanie bolo pri pridávaní v určitých situáciách zo všetkých najrýchlejšie. Ak pred pridávaním vieme celkový počet prvkov, tak sa vytvorí dostatočne veľká tabuľka, nebude potreba tvoriť novú, a prvky sa do nej rýchlo

uložia. V najhoršom prípade, ak by sme začínali s najmenšou možnou tabuľkou, ktorú by sme postupne zväčšovali, by bola rýchlosť pridávania približne na rovnakej úrovni, ako pri červeno-čiernom strome. Avšak to je len najhorší možný prípad, takže celkovo je moje hashovanie zo všetkých testovaných dátových štruktúr najrýchlejšie. Prevzaté hashovanie nemalo zväčšujúcu sa tabuľku, takže rýchlosť vkladania závisela aj od toho, akú veľkú ju nastavíme. Ak by sme dopredu nevedeli počet pridávaných prvkov, tak použiť prevzaté hashovanie sa neoplatí, keďže nám môže dôjsť miesto v tabuľke. Celkovo najvýhodnejšie je použiť buď červeno-čierny strom, alebo moje hashovanie, záleží to aj od toho, či vieme, koľko prvkov chceme vložiť. Ak to vieme, výhodnejšie je použiť hash, ak nevieme, rozdiel bude nepatrný, ale hash bude stále mierne rýchlejší.

Vkladanie	Unbalanced Tree	AVL Tree	Red-black Tree	Môj hash		Prevzatý hash	Veľkosti vstupov
				so zväčšovaním	bez zväčšovania		
Insert (10 000) postupne	0.45	0.015	0.008	0.004	0.002	0.003	<1; 10 000>
Insert (10 000) random	0.008	0.015	0.006	0.006	0.003	0.004	<1; 100 000>
Insert (25 000) postupne	2.5	0.035	0.02	0.017	0.005	0.006	<1; 25 000>
Insert (25 000) random	0.02	0.04	0.02	0.02	0.007	0.007	<1; 100 000>
Insert (50 000) postupne	netestoval som	0.077	0.04	0.035	0.012	0.01	<1; 50 000>
Insert (50 000) random	0.045	0.073	0.036	0.03	0.012	4.5	<1; 1 000 000>
Insert (100 000) postupne	netestoval som	0.15	0.08	0.07	0.023	0.025	<1; 100 000>
Insert (100 000) random	0.096	0.14	0.07	0.08	0.03	netestoval som	<1; 1 000 000>

Rýchlosť vyhľadávania prvkov v dátových štruktúrach

V nevyváženom strome je vyhľadávanie prvkov, ak ich vkladáme ako postupnosť pomalé, keďže v tom prípade sa nejedná o binárny strom, ale o spájaný zoznam. Ak v ňom vyhľadávame náhodné prvky, tak čas trvania je o poznanie rýchlejší, ale stále najpomalší zo všetkých testovaných dátových štruktúr. Vyhľadávanie v AVL strome je najrýchlejšie zo všetkých binárnych stromov, a vyniká najmä, ak hľadáme vloženú postupnosť čísel, v tomto prípade je takmer dvakrát rýchlejšie ako červeno-čierny strom. V červeno-čiernom strome trvá vyhľadávanie menej ako v nevyváženom, ale viacej ako v AVL strome. Rýchlosť hľadania náhodných prvkov je podobná rýchlosti v AVL strome a rýchlosť vyhľadania postupnosti je o poznanie pomalšia, ale zas červeno-čierny strom sa osvedčí najmä pri pridávaní prvkov. Obe hashovacie tabuľky sú o poznanie rýchlejšie ako vyhľadávacie binárne stromy, keďže v nich je v najlepšom prípade rýchlosť vyhľadania takmer okamžitá, lebo hľadáme podľa indexu v poli. Avšak prevzatý hash bol pri hľadaní náhodných prvkov od istého množstva pomalý, keďže použitý kód bol jednoduchý a neriešil zväčšovanie tabuľky, čo by v tomto prípade pomohlo. Celkovo najvýhodnejšie je použiť môj hash, keďže zložitosť hľadania je O(1), a prvky sa vo väčšine prípadov nájdu okamžite. Ak by sme chceli na vyhľadávanie použiť strom, tak najideálnejší je AVL, ktorý vyniká najmä pri hľadaní postupností.

Vyhľadávanie	Unbalanced Tree	AVL Tree	Red-black Tree	Môj hash	Prevzatý hash	Veľkosti vstupov
Search (10 000) postupne	0.33	0.003	0.006	0.001	0.001	<1; 10 000>
Search (10 000) random	0.006	0.005	0.005	0.002	0.001	<1; 100 000>
Search (25 000) postupne	2.1	0.009	0.015	0.002	0.002	<1; 25 000>
Search (25 000) random	0.015	0.011	0.012	0.002	0.004	<1; 100 000>
Search (50 000) postupne	netestoval som	0.019	0.03	0.003	0.005	<1; 50 000>
Search (50 000) random	0.03	0.023	0.025	0.006	1.75	<1; 1 000 000>
Search (100 000) postupne	netestoval som	0.035	0.06	0.007	0.007	<1; 100 000>
Search (100 000) random	0.063	0.045	0.05	0.01	netestoval som	<1; 1 000 000>

Počet operácii pri vkladaní prvkov

Ďalšia vec, čo som testoval, bol počet vykonaných operácii. Nevyvážený strom bol pri vkladaní číselnej postupnosti veľmi neefektívny, keďže celkový počet vykonaných operácii bol už pri 10,000 prvkoch vyše 50 miliónov. Avšak vykonané operácie pri náhodných vstupoch boli nízke. Pri AVL strome bol počet vykonaných operácii vyšší ako pri strome nevyváženom, keďže je treba vykonať aj rotácie, ktoré celkový výsledok zvyšujú. Prekvapivo, červeno-čierny strom vykoná nižší počet operácii ako nevyvážený strom, aj keď v ňom dochádza k rotáciám. Pri hashovaní záleží aj od toho, či je potrebné tabuľku zväčšovať, alebo nie. Ak nie, tak počet vykonaných operácii je veľmi nízky, a to som doň zarátal aj operácie, ktoré sa vykonajú pri samotnej tvorbe tabuľky. V najhoršom možnom prípade bude počet operácii o poznanie vyšší, ale stále bude nižší ako väčšina ostatných dátových štruktúr. Pri menších vstupoch bude stále nižší ako počet operácii pri binárnych stromoch, ale neskôr sa zvýši, keďže vytvoriť novú tabuľku a presunúť všetky prvky do nej je náročné na počet operácii. Celkovo je najvýhodnejšie použiť červeno-čierny strom, ak nevieme presný počet prvkov, no vieme, že ich bude veľa. Ak počet prvkov poznáme, tak najmenej operácii zaberie hashovanie.

Vkladanie	Unbalanced Tree	AVL Tree	Red-black Tree	Môj hash	
				so zväčšovaním	bez zväčšovania
Insert (10 000) postupne	50 005 000	267 000	231 000	102 000	42 000
Insert (10 000) random	165 000	247 000	132 000	105 000	43 000
Insert (25 000) postupne	312.5 mil	734 000	643 000	375 000	105 000
Insert (25 000) random	465 000	655 000	352 000	387 000	107 000
Insert (50 000) postupne	netestoval som	1 569 000	1 386 000	748 000	210 000
Insert (50 000) random	1 000 000	1 334 000	716 000	785 000	224 000
Insert (100 000) postupne	netestoval som	3 337 000	2 972 000	1 492 000	418 000
Insert (100 000) random	2 165 000	2 677 000	1 437 000	1 662 000	502 000

Počet operácii pri vyhľadávaní prvkov

Pri vyhľadávaní je počet vykonaných operácii pri nevyvážených stromoch najvyšší zo všetkých dátových štruktúr, lebo je potrebné prejsť cez veľa prvkov, aby sme sa dostali na ten hľadaný. Pri AVL strome je počet operácii najnižší spomedzi všetkých binárnych vyhľadávacích stromov. Rozdiely medzi hľadaním postupnosti a náhodných čísel nie sú také veľké ako pri červeno-čiernom strome, kde je počet vykonaných operácii len o niečo vyšší od AVL stromu pri náhodných vstupoch, a o niekoľko tisíc vyšší pri postupnostiach. Pri hashovaní je počet vykonaných operácii pri postupnostiach dvojnásobne väčší, ako počet vložených prvkov, keďže údaje budú v tabuľke uložené ako pole a nedôjde ku kolíziám. Ak boli vložené náhodné prvky, tak počet operácii bude približne rovnaký. Celkovo je to najmenej náročná dátová štruktúra, keď hovoríme o počte operácii, takže by bolo najviac výhodné použiť práve ju.

Vyhľadávanie	Unbalanced Tree	AVL Tree	Red-black Tree	Môj hash
Search (10 000) postupne	49 995 000	113 000	119 000	20 000
Search (10 000) random	153 300	113 000	113 000	21 000
Search (25 000) postupne	312.5 mil	317 000	327 000	50 000
Search (25 000) random	420 000	306 000	307 000	49 000
Search (50 000) postupne	netestoval som	684 000	705 000	100 000
Search (50 000) random	900 000	637 000	637 000	99 000
Search (100 000) postupne	netestoval som	1 469 000	1 509 000	200 000
Search (100 000) random	1 835 000	1 305 000	1 307 000	199 000

Záver

Vo svojom riešení som implementoval nevyvážený binárny vyhľadávací strom, AVL strom a hashovanie, kde sa kolízie riešili reťazením pomocou spájaných zoznamov. Okrem toho som prevzal červeno-čierny strom a hashovanie, kde sú kolízie riešené otvoreným adresovaním. Potom som všetky dátové štruktúry podrobne otestoval a získané výsledky som zaznamenával a porovnával medzi sebou, aby som sa dozvedel, ktorá z nich je výhodná v akej situácii. Obe hash tabuľky boli vo väčšine prípadov najrýchlejšie a počet operácii bol taktiež nižší ako u iných dátových štruktúr, takže je najviac výhodné použiť tie, keďže pri vkladaní a vyhľadávaní nemusíme prechádzať cez celú štruktúru, ako to je pri binárnych vyhľadávacích stromoch, ale takmer okamžite sa dostaneme na požadované miesto. Z dvoch hash tabuliek je moja lepšia, keďže ju je možné aj zväčšiť, ak by sme náhodou potrebovali vložiť ďalšie prvky. Nevyvážený binárny strom sa neoplatí používať, oveľa lepšie je využiť niektorú z jeho vyvážených verzii. Je ťažké povedať, ktorá z dvoch implementovaných verzii je lepšia, keďže každá má svoje výhody a aj nevýhody. Vďaka testovaniu som prišiel na to, že červeno-čierny strom sa viacej oplatí použiť pri vkladaní prvkov a AVL strom je zas efektívnejší pri vyhľadávaní.