VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HODNOCENÍ VYBRANÝCH METOD VYHLEDÁVÁNÍ VE STROMOVÝCH STRUKTURÁCH

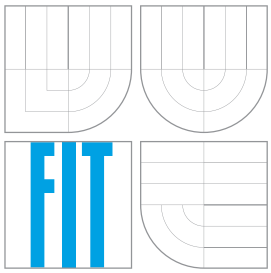
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VÍT TŘÍSKA

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HODNOCENÍ VYBRANÝCH METOD VYHLEDÁVÁNÍ VE STROMOVÝCH STRUKTURÁCH

EVALUTAION OF CHOSEN TREE SEARCH METHODS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VÍT TŘÍSKA

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. JAN MAXMILIÁN HONZÍK, CSc.

BRNO 2008

Abstrakt

Úloha vyhledávat je široce rozšířená operace ve všech jejích možných podobách. Současný vývoj ukazuje na vzrůstající význam efektivních vyhledávacích metod. Proto tato práce se zabývá hodnocením nejrychlejších vyhledávacích metod současnosti, a to AVL stromem, červeno-černým stromem, rozvinutým stromem a přeskakujícím seznamem. Kromě toho se snaží doporučit je k jejich správnému použití. Doplňkovým cílem této práce je vytvořit učební nástroj (studijní pomůcku), která by ulehčila lepšímu porozumění úskalí každé individuální vyhledávací metody.

Klíčová slova

Vyhledávací algoritmy, AVL strom, červeno-černý strom, rozvinutý strom, přeskakující seznam, rotace, knihovna metod vyhledávání, čítače a časovače, RDTSC, wxWidgets, vykreslování stromu, demonstrační program

Abstract

A function to search is widely used operation in all possible forms. Recent developments show increasing importance of far more effective search methods. That is why this work deals with an evaluation of the fastest search methods at present time, namely AVL tree, red-black tree, splay tree and skip list. Moreover, it tries to recommend them for their correct usage. A supplementary goal of this thesis is to create a learning tool (study aid), which would facilitate better understanding of pitfalls of each individual search method.

Keywords

Search algorithms, AVL tree, red-black tree, splay tree, skip list, rotations, library of search methods, ticks and counters, RDTSC, wxWidgets, drawing tree, demonstrational program

Citace

Vít Tříska: Hodnocení vybraných metod vyhledávání ve stromových strukturách, bakalářská práce, Brno, FIT VUT v Brně, 2008

Hodnocení vybraných metod vyhledávání ve stromových strukturách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Jana Maxmiliána Honzíka, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vít Tříska
9. května 2008

Poděkování

Děkuji tímto vedoucímu své bakalářské práce, panu prof. Ing. Janu Maxmiliánu Honzíkovi, CSc., nejenom za vedení a cenné rady, ale také za jeho připomínky a podnětné návrhy na různá rozšíření či zlepšení v celém období činnosti na této práci. Dále děkuji panu Dr. Ing. Petru Peringerovi za rady k tvorbě testovacího programu a následné odpovědi k řešení drobných nesrovnalostí, na které jsem v průběhu práce narazil. Rád bych touto formou poděkoval i ostatním osobám, které si nepřály být zde jmenovány, a laskavému čtenáři této práce, který uzná za vhodné se touto prací zabývat a omluví případné nedostatky, které nelze nikdy vyloučit.

Obsah

Kapitola 1

Úvod

Vyhledávací algoritmy jsou jednou z nejvýznamnějších oblastí tvorby algoritmů i programování už od jejich samotných počátků. Ačkoli se od těchto začátků v programování a částečně i algoritmizaci ledacos změnilo, významnost vyhledávání zůstává stále zachována. Proto se hledají a objevují různé vyhledávací algoritmy, které mají různou principiální složitost. Vzhledem k tomu, že je v současnosti tlak na užívání čím dál tím více efektivnějších vyhledávacích metod, práce se zabývá hodnocením rychlými vyhledávacími metodami, za které lze považovat AVL strom (AVL tree), červeno-černý strom (red-black tree), rozvinutý strom (splay tree) a přeskakující seznam (skip list). Tyto metody vznikaly na sobě nezávisle a postupně. Nemusí být tak zcela zřejmé, kdy jakou vyhledávácí metodu je vhodné užít. Proto cílem této práce je tyto vyhledávací metody vzájemně porovnat a navrhnout doporučení k jejich užití. Toto porovnání má však být podepřeno nejen všeobecnými znaky jednotlivých metod, ale i experimentálními poznatky. To však nezbytně vyžaduje jejich implementaci a implementaci testovacího programu, které jsou také předmětem práce. Vzhledem k tomu, že výsledek práce může být dále pomůckou při studiu těchto metod, je vyžadována také tvorba demonstračního programu pro grafické znázornění chování rozebíraných metod.

Textová část práce se skládá z pěti částí. První část, tradičně nazývaná „Úvod“ (1), stručně uvádí do řešené problematiky, seznamuje se zadáním a cílem práce. Druhá část nazvaná „Vybrané vyhledávací metody“ (2) seznamuje čtenáře se zadanými vyhledávacími metodami jak po teoretické, tak i praktické části a seznamuje s výsledkem implementace těchto metod. Třetí část pojmenovaná „Hodnocení vyhledávacích metod“ (3) se zabývá lehce složitostí, návrhem hodnocení, způsobem provedení realizace měření, výsledky z měření a jejich vyhodnocením. Čtvrtá část „Demonstrační program“ (4) se zabývá tvorbou demonstrační aplikace, možnostmi této aplikace, algoritmy vykreslování jednotlivých struktur a vlastním zhodnocením této části. Poslední část „Závěr“ (5) shrnuje poznatky a výsledky z provedené práce.

Kapitola 2

Vybrané vyhledávací metody

2.1 Principy vybraných metod vyhledávání (teoretická část)

2.1.1 AVL strom (AVL tree)

AVL strom[1] je samovyvažující binární vyhledávací strom. Jeho algoritmus byl publikován v roce 1962 autory G. M. Adelson-Velsky a E. M. Landis v [1]. Ti dokázali, že takto navržený strom je vyšší maximálně o 45 % proti dokonale (váhově, absolutně) vyváženému stromu vytvořeného nad stejným počtem uzlů. Význam tohoto stromu spočívá v rychlé činnosti operací jako jsou vkládání, hledání, mazání, které se odehrávají v logaritmickém čase i se zajištěním samovyvažovatelnosti stromu. Strom je pojmenován jako AVL strom podle počátečních písmen svých objevitelů (**A**delson-**V**elsky, **L**andis).

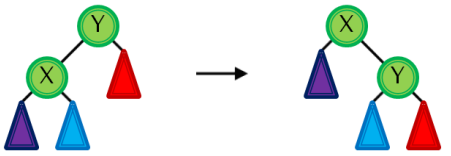
Základem AVL stromu je binární stromová struktura, o které platí, že každý uzel má nejvýše dva následníky. Dále má vlastnosti vyhledávacího stromu. Uzly s menší hodnotou klíče než kořen jsou v levém podstromu. Jinak jsou uzly větší jak kořen a nachází se napravo od kořene. Tyto vlastnosti platí pro každý uzel stromu a jsou charakteristické nejen pro AVL strom. Hlavní vlastnost typická pro tento strom tkví v tom, že je výškově vyvažován. To znamená, že pro všechny uzly stromu platí, že výška levého podstromu se rovná výšce pravého podstromu, nebo se liší právě o 1. Někdy je tato vlastnost vyjádřena jinými slovy následovně: *Délka nejdelší větve levého a pravého podstromu se liší nejvýše o 1.* AVL strom se tedy řadí do binárních stromů, vyhledávacích stromů a má svoji výškově vyvažovací charakteristiku.

Protože AVL strom je vždy vyvažován, nemůže nikdy zdegenerovat. Logaritmus velikosti reprezentované množiny udává minimální výšku stromu[2]. Pro uchování si žádané výšky stromu nám slouží čtyři rotace, které ve svém pojmenování kombinují levý (left) a pravý (right) směr. Jsou jimi *LL rotace* (*rotace zleva*), *RR rotace* (*rotace zprava*), *DLR rotace* (*rotace zleva zprava*) a *DRL rotace* (*rotace zprava zleva*). LL rotace a RR rotace jsou jednoduché rotace. DLR a DRL rotace jsou dvojité rotace. Jednotlivé případy jejich efektu činnosti jsou uvedeny na obrázku 2.1 (str. 5).

Každý uzel stromu má možnost zjistit rozdíl výšek pravého a levého podstromu. Tento

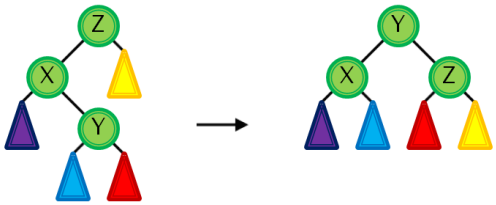
[1] Při zpracování bylo čerpáno z těchto studijních materiálů: [4, 4. monografie], [8, strana 121–132], [10, strana 11–17], [18] a [7]

[2] Platí obecně pro každý binární strom: Nechť l (level) je minimální počet úrovní stromu a n (node) je maximální počet uzlů stromu, které se vejdou do daného počtu úrovní. Potom mezi nimi platí vztahy: $l = \lceil \log_2(n+1) \rceil$, neboli $n = 2^{\lceil l \rceil} - 1$.
Př.: Kolik minimálně úrovní bude mít strom pro uložení 12 uzlů? $l = \lceil \log_2(12+1) \rceil \doteq \lceil 3,7 \rceil = 4$
Př.: Kolik maximálně uzlů se vejde do 4 úrovní stromu? $n = 2^{\lceil 4 \rceil} - 1 = 16 - 1 = 15$

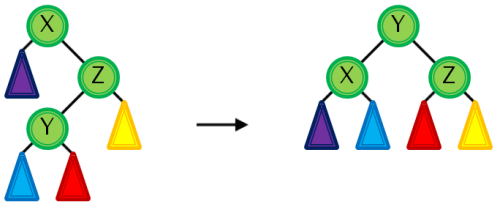


(a) LL rotace; rotace doprava, rotace zig

(b) RR rotace; rotace doleva; rotace zag

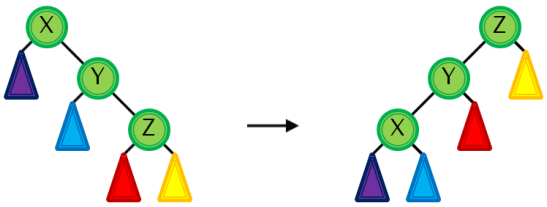


(c) DLR rotace; –; rotace zig-zag



(d) DRL rotace; –; rotace zag-zig

(e) –; –; rotace zig-zig



(f) –; –; rotace zag-zag

Obrázek 2.1: Přehled rotací v *AVL stromu* (*AVL*), *červeno-černém stromu* (*RBT*) a *rozvinutém stromu* (*SPT*)

Pojmenování rotací je uvedeno v pořadí *AVL* – *RBT* – *SPT*, pomlčka (–) znamená, že tuto rotaci metoda nepoužívá. Všimněme si toho, že všechny stromy používají stejné rotace, které se jen různě nazývají. Např. LL rotace v *AVL* stromu (někdy nazývána jako rotace zleva, nikoli doleva) je významově totožná co rotace doprava v *RBT*. Také je si dobré uvědomit, že všechny dvojité rotace, jimiž jsou 2.1(c), 2.1(d), 2.1(e) a 2.1(f), jsou odvozeny z jednoduchých rotací 2.1(a) a 2.1(b). Provedeme-li dvakrát správně zvolené jednoduché rotace nad správným uzlem a ve správném pořadí, obdržíme stejný výsledek jako při použití přímo některé dvojité rotace.

rozdíl se nazývá *vyvažovací faktor* nebo někdy jednoduše jako váha uzlu. Dodržuje se tato konvence v označování výškové vyváženosti uzlů: Je-li levý a pravý podstrom nějakého uzlu stejně vysoký, je aktuální vyvažovací faktor tohoto uzlu roven 0. Pokud se přidá k takovému uzlu jeden uzel nalevo či napravo, váhový rozdíl bude činit -1, pokud je uzel vlevo výškově větší (uzel přidán nalevo), nebo $+1$, pokud je uzel vpravo výškově větší (uzel přidán napravo). Pokud se k takto nakloněné straně přidá ještě další uzel, stane se výška o $-/+2$ uzly nevyvážená, což už porušuje pravidlo výškové vyváženosti a volá se příslušná rotace na její rozvážení. Uzel, ve kterém došlo k poruše vyváženosti $(-/+2)$, nazýváme kritickým uzlem. Naopak, pokud se přidá uzel ke straně opačné, naklonění se může vyrovnat na 0. Toto vyvažování je třeba ve stromě kontrolovat a při poruše napravovat nad každým uzlem stromu. Je nezbytné, aby každý uzel měl hodnotu vyvažovacího faktoru z množiny $\{-1, 0, +1\}$, jiné stavy porušují vyváženost stromu a musí se řešit. Dodejme, že k poruše vyváženosti může dojít jen při změně počtů uzlů ve stromu, neboli při operacích jako je vkládání a mazání.

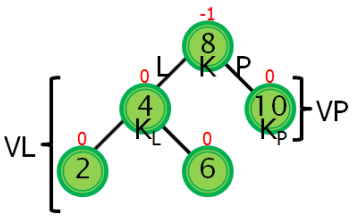
Přidáme-li nebo odebereme-li někde uzel, je nutno zjistit nově vyvažovací faktor nad každým uzlem, jehož změna výšky stromu se týká, neboli překontrolovat vyvažovací faktor pro každý uzel směrem ke kořenu stromu. Předpokládejme, že nový uzel přidáme do levého podstromu, čímž se jeho výška zvětší o 1. Vyvážení uzlu s počátečním stavem uváděném ve výčtu se změní následovně:

VL $<$ VP: Před přidáním uzlu byl levý podstrom o výšce VL o 1 uzel nižší. Přidáním uzlu vlevo bude výška VL $=$ VP. Dojde tedy k vyvážení uzlu, protože rozdíl výšek je 0. O vyvažování stromu nemůže být řeč.

VL $=$ VP: Před přidáním uzlu byly oba podstromy stejně vysoké. Přidáním uzlu vlevo bude výška VL $>$ VP. Dojde tedy k naklonění na levou stranu, ale protože rozdíl výšek je jen o 1, není třeba provádět vyvažování stromu.

VL $>$ VP: Před přidáním uzlu byl levý podstrom o 1 uzel vyšší. Přidáním uzlu vlevo bude výška VL $>$ VP už o 2 uzly větší. Dojde tedy k narušení vyvážení stromu z levé strany. Proto je nutno volat vhodnou rotaci, která by nevyváženost odstranila.

Výběr vhodné rotace je závislý na tom, z které strany je v důsledku přidání/odebrání podstrom nevyváženého uzlu těžší. Je-li těžší vlevo, vybíráme z rotací LL a DLR, jinak RR a DRL. Další rozhodnutí, kterou konkrétní rotaci z určené dvojice vybrat, je dle naklonění přímého potomka (kořene podstromu) v tom podstromu, který je těžší. Je-li tento potomek nakloněn na stejnou stranu jako nevyvážený uzel nebo je právě vyvážen (případ vznikající u mazání), provedeme jednoduchou rotaci (rotace LL, RR), je-li opačně nakloněn, použijeme dvojitou rotaci (rotace DLR, DRL). Rotace není v podstatě nic jiného než cyklická záměna ukazatelů (dle přehledu rotací uvedených v 2.1, str. 5).



Obrázek 2.2: Ukázka vyváženého AVL stromu

Uveďme krátký příklad: Označme kořen K, levý podstrom L, jeho výšku VL, kořen levého podstromu K_L, pravý podstrom P, jeho výšku VP a kořen pravého podstromu K_P (viz obr. 2.2), kde pro vyšší názornost byly kromě uvedeného značení vypočteny i aktuální faktory vyvážení. Přidáme-li např. uzel s klíčem 9 nebo 11, k poruše vyvážení žádného uzlu nedojde. Uzel s klíčem 10 se stane jen nakloněným $(-/+1)$ a uzel s klíčem 8 se stane lépe vyváženým (0). Přidáme-li např. uzel s klíčem 5 nebo 7, dojde k naklonění uzlu s klíčem 6 $(-/+1)$ i 4 $(+1)$ a k poruše vyvážení uzlu s klíčem 8 (-2). K opětovnému vyvážení použijeme v souladu s dříve uvedeném rotaci DLR (v K nastala porucha vyvážení, K_L je $+1$. Přidáme-li např. uzel s klíčem 1 nebo 3,

dojde k naklonění uzlu s klíčem 2 ($-/+1$) i 4 (-1) a k poruše vyvážení uzlu s klíčem 8 (-2). K opětovnému vyvážení použijeme v souladu s dříve uvedeném rotaci LL (v K nastala porucha vyvážení, K_L je -1).

Mezi nejběžnější operace patří vyhledávání, vkládání a rušení uzlu stromu. Zatímco u vyhledávání se výška stromu nikdy nemění a probíhá zcela stejně jako u binárního vyhledávacího stromu, při potřebě přidat nebo zrušit uzel v AVL stromu se musí sledovat vyvažovací faktor uzlu. K jeho zjišťování jsou dva přístupy. Buď si ho vždy spočítat a typicky používat tak rekurzivní implementaci jeho zjišťování, nebo si jej značit vždy do každého uzlu a při změně uspořádání (rotaci) neopomnět jej aktualizovat (nerekurzivní, tzv. iterativní přístup). Rekurzivní metoda bývá pro implementaci jednodušší. Je však o něco pomalejší, protože se provádí vždy rekurzivní volání nad každým uzlem pro určení potřebných výšek stromu. Dalším možným problém rekurze může být to, že všechny lokální proměnné a parametry funkcí se automaticky ukládají do části paměti nazývané zásobník. O zásobníku předem nemůžeme zjistit (alespoň v jazyce C, jeho volba zdůvodněna v části 2.2, str. 25) jeho velikost, protože zásobník řídí a stará se o něj operační systém. Zmíněné ukládání se děje při každém rekurzivním zanoření. Může tak dojít k zastavení chodu programu z důvodu nedostatku velikosti zásobníku. Naštěstí při jeho použití však toto částečně skryté nebezpečí nehrozí, protože strom je velice dobře vyvažován. I nejdelší cesta, která určuje potřebný počet zanoření, je ve svém počtu procházených uzlů přijatelná. Ověřme:

> Zjistěme nejdříve, jaký bývá maximální počet rekurzivního zanoření na současných běžně dostupných strojích a operačních systémech (Windows, Linux, FreeBSD). Nezbývá než prakticky ověřit. Ze sestaveného testu s rekurzivním voláním procedury se 2 parametry typu `int` (8 bajtů) a 1 lokální proměnnou pro jednoduchost také typu `int` (4 bajty) se zjistilo, že nejméně 50 000 krát lze provést rekurzivní zanoření na všech běžně dostupných operačních systémech[3]. Uvážíme-li, že adresní prostor je na 32-bitových systémech shora omezen na 4 GB (2^{32} bajtů) paměti a velikost prostého stromového uzlu bývá nejméně 12 bajtů[4], zjistíme, že lze teoreticky do této paměti uložit $(\frac{2^{32}}{12})$ uzlů při zanedbání veškerých potřeb na paměť ostatních aplikací i vlastního operačního systému (a při odhlédnutí od problematiky zarovnaného přístupu k paměti a chráněné paměti). Minimální počet úrovní stromu k uložení tohoto počtu je pak dán vztahem $l = \lceil \log_2(n+1) \rceil$, což je $l = \lceil \log_2(\frac{2^{32}}{12}+1) \rceil \doteq \lceil 28.4 \rceil = 29$. I kdybychom vzali dvojnásobek (víme že je odvozeno, že výška je max o 45 % větší), tato hodnota je proti experimentálně zjištěnému maximálnímu počtu zanoření nesrovnatelně menší. Lze tedy rekurzi v implementaci AVL stromu bez obav použít.

Po dlouhém zvažování, zda se vydat rekurzivním či iterativním směrem, byla nakonec pro jednotnost všech metod vybrána iterativní implementace. K rozhodnutí navíc přispěla i skutečnost, že u rekurze lze očekávat, že bude pomalejší. Protože cílem práce je provádět

[3] Konkrétní naměřené hodnoty maximálního rekurzivního zanoření jsou následující: vlastní pracovní stanice (3 GB RAM): MS Windows Vista – 65088, (512 MB RAM) MS Windows 2000/XP – 65093, Linux SUSE – 174562, Linux CentOS – 261974, školní servery Merlin, Eva (4 GB RAM): Linux CentOS – 524022, FreeBSD – 2096928. (Spatřena zajímavá vlastnost, že velikost zásobníku u MS nezávisí na velikosti operační paměti.) Jednoduchý testovací program taktéž přikládám na CD uvedeném v příloze.

[4] Nejmenší struktura uzlu stromu vyžaduje znát alespoň svůj vyhledávací klíč a 2 ukazatele na levého a pravého potomka. Za klíč bývá nejčastěji používán typ `int`. V jazyce C bývá velikost typu `int` 4 bajty, taktéž typ ukazatel bývá nejčastěji 4 bajty. Celkově je nejmenší velikost uzlu rovna 12 bajtům, pokud nebudeme uvažovat zarovnávání paměti.

měření na efektivitu, bude dále v textu uváděn iterativní popis vyvažování AVL (a provedena iterativní implementace AVL stromu).

Zaměřme se na maximální počty vyvažování stromu. Lze zjistit, že zatímco u vkládání vždy platí, že kterýkoli případ myslitelné rotace provede nápravu porušení vyvážení a zachování stejné výšky stromu jako před vložením uzlu, u mazání toto neplatí. Proto se u přidávání uzlu k nápravě vyváženosti volá nejvýše jedna rotace, zatímco u mazání uzlu lze při jistém uspořádání uzlů stromu volat tolik rotací, kolik je uzlů na cestě ke smazanému uzlu. Blíže jsou všechny možné situace rozkresleny na příslušných minimálních stromech pro vkládání 2.3 (str. 9) a mazání 2.4 (str. 10) v případech použití LL nebo DLR rotace. RR a DRL rotace jsou jen zrcadlově obrácené, a proto v přehledu nejsou uvedeny. Všimněte si, jak se mění výška stromu před a po provedení rotace při vkládání/odstranění uzlu (potvrzení platnosti tvrzení o maximálním počtu rotací) a jak se mění vyvažovací faktory jednotlivých uzlů (především nutno znát pro implementaci iterativním způsobem). Je vhodné se nad nimi zamyslet obzvláště proto, protože na naší fakultě existují v mnohých studijních materiálech i pracích tohoto charakteru různé chybné iterativní implementace AVL stromu, ve kterých se nejedná pouze o opomenutí správného přepisu vyvažovacího faktoru (jak je často nesprávně uváděno v operaci vkládání), ale i o fundamentální nedostatky ošetření všech možných případů konfigurace stromů, které mohou nastat (jak často chybí v operaci mazání uzlu).

U AVL stromů je empiricky zjištěno, že v průměru nastává jedno vyvažování po každých 2 přidaných nových uzlech. Při tom jsou jednoduché a dvojité rotace stejně pravděpodobné. Dochází-li k rušení uzlu, bylo empiricky zjištěno, že rotace nastává zhruba na každé páté zrušení. Ale díky své náročnosti je možno považovat rušení uzlů v AVL stromech za stejně složité jako přidávání (viz [5, strana 156 a 158]).

Z předešlého vyplývá, že práce s AVL stromem je náročnější pro vkládání či odstraňování uzlů, nežli práce nad obyčejným binárním vyhledávacím stromem. Odměnou za toto úsilí by měla být zkrácená doba vyhledávání. Srovnáme-li AVL strom s dokonale (váhově) vyváženým stromem[5], nejsou ještě uvedené rotace tak často volány, protože je odstraněna nutnost vždy vyvažovat uzly při jakékoli změně jejich počtu. Protože strom je jen o 45 % vyšší než dokonale vyvážený strom, má tato vyhledávací metoda dobrý předpoklad k rychlému vyhledávání. Proto by mělo být vhodné ji využívat především tam, kde vyhledávání převažuje nad přidáváním nebo rušením jejich uzlů.

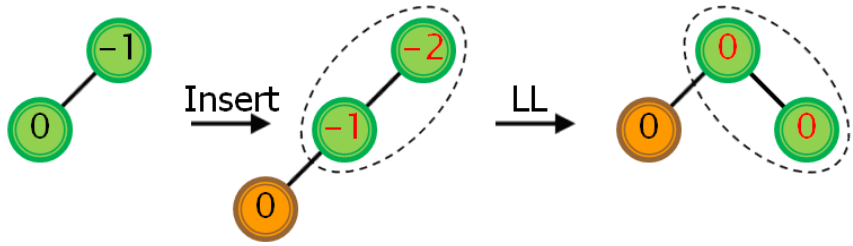
2.1.2 Červeno-černý strom (red-black tree)

Červeno-černý strom[6] je samovyvažující binární strom, který byl vynalezen v roce 1972. Jeho autorem je Rudolf Bayer, který jej nazval symetrickým binárním B-stromem („symmetric binary B-trees“, viz [2]). Toto jméno se však neuchytilo. Jeho novodobé pojmenování získal až z práce Leonida J. Guibase a Roberta Sedgewicka z roku 1978 (viz [6]). Název této stromové struktury je patrně odvozen ze skutečnosti, že každý uzel má svůj jen dvouhodnotový barevný atribut – červenou (red) nebo černou (black) barvu.

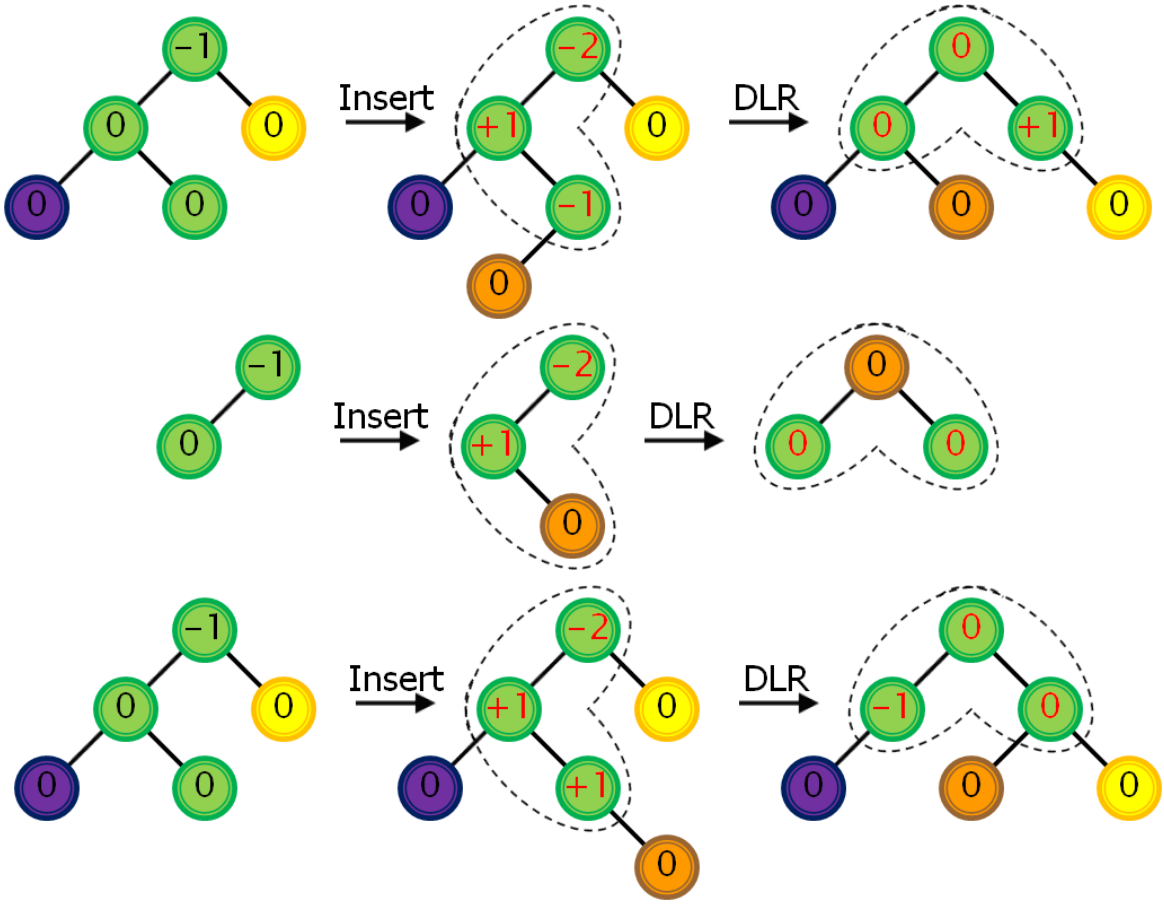
Červeno-černý strom si zachovává všechny vlastnosti binárního vyhledávacího stromu. Jeho nejbližším příbuzným je 2-3-4 strom, z kterého pravděpodobně vychází, ale narozdíl od 2-3-4 stromu má jen jeden typ uzlů. Díky tomu není jeho implementace tak těžkopádná. Jak

[5] *Binární strom je váhově vyvážený, když pro všechny jeho uzly platí, že počty uzlů jejich levého podstromu a pravého podstromu se rovnají, nebo se liší právě o 1.*

[6] Při zpracování bylo čerpáno z těchto studijních materiálů: [4, 6. monografie], [3] a [24]



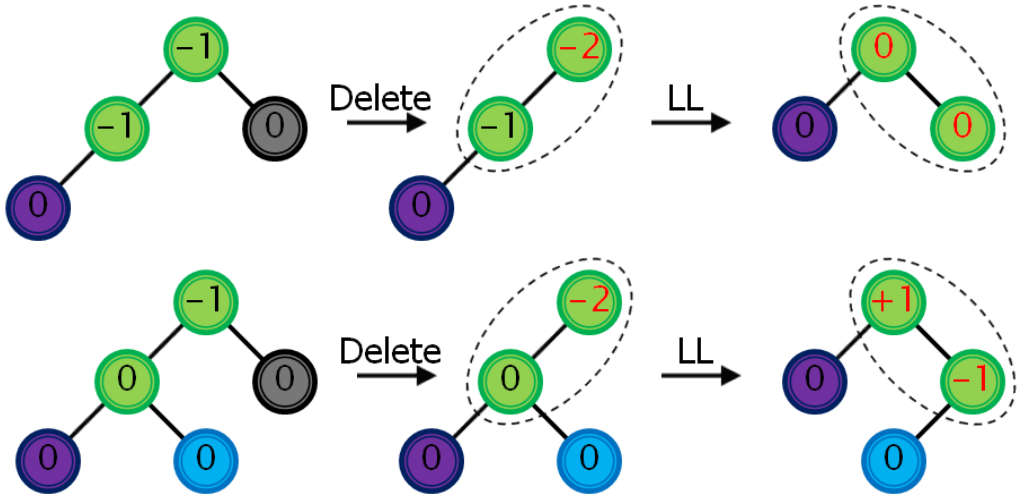
(a) Možné případy uspořádání AVL stromu při použití LL rotace



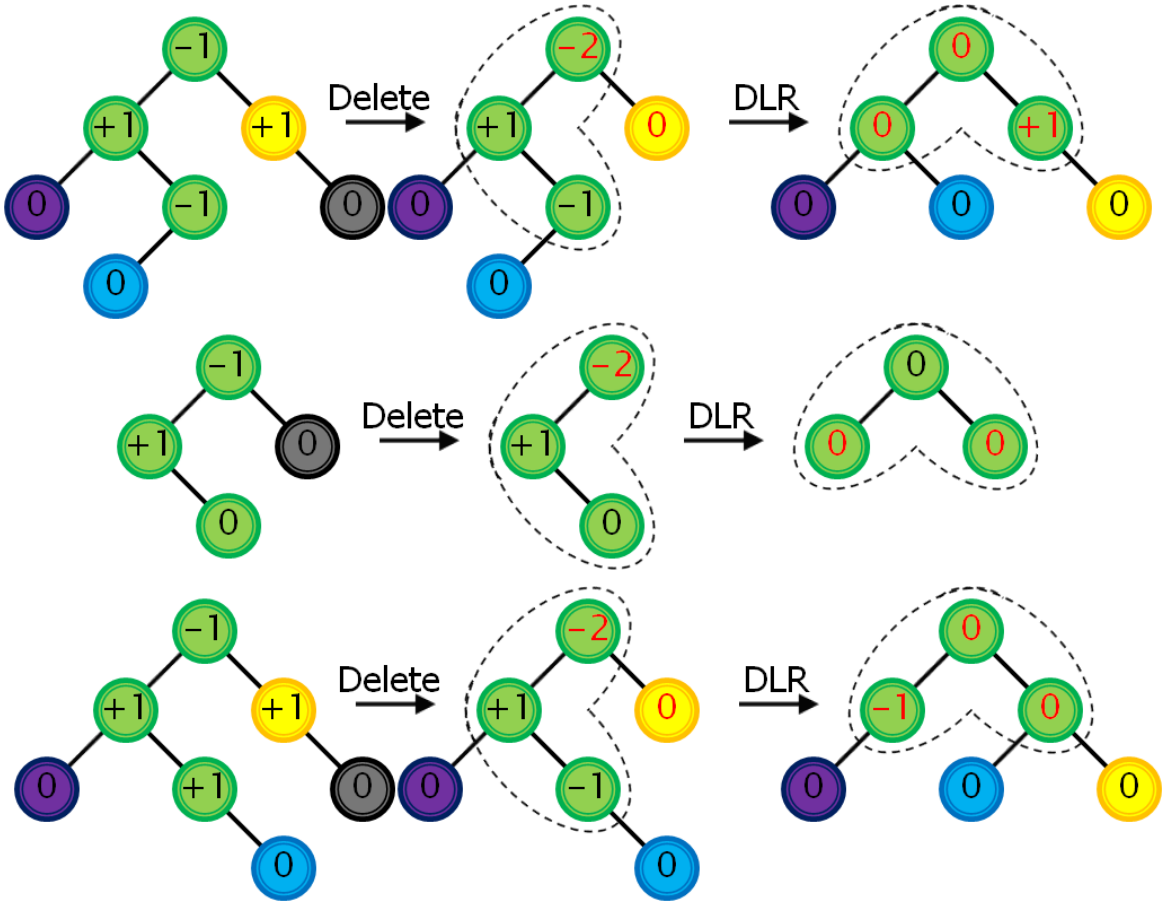
(b) Možné případy uspořádání AVL stromu při použití DLR rotace

Obrázek 2.3: Možné případy uspořádání AVL stromu při *vložení uzlu* vedoucí k nutnosti provést vyvážení za pomoci LL nebo DLR rotace (RR a DRL rotace jsou jen zrcadlově obráceny)

U vkládání se výška stromu před přidáním uzlu i po jeho přidání se zavoláním příslušné rotace celkově nezmění. Proto není třeba procházet vyvažovací faktor od kritického uzlu směrem ke kořenu stromu a platí, že u vkládání je volána nejvýše jedna rotace.



(a) Možné případy uspořádání AVL stromu při použití LL rotace



(b) Možné případy uspořádání AVL stromu při použití DLR rotace

Obrázek 2.4: Možné případy uspořádání AVL stromu při *mazání uzlu* vedoucí k nutnosti provést vyvážení za pomoci LL nebo DLR rotace (RR a DRL rotace jsou jen zrcadlově obráceny)

U mazání se odebráním uzlu většinou sníží výška stromu. Je tedy nutné od kritického uzlu směrem ke kořenu stromu zkontrolovat a upravit vyvažovací faktory, zda nebude porušeno vyvážení. Pokud ano, provede se vyvážení a pokud se opět sníží výška, pokračuje se ke kořenu stromu, jinak se končí. Proto u mazání platí, že se provede nejvýše takový počet rotací, kolik je uzlů na cestě z kořene k odebranému uzlu, neboli nejvýše $O(\log n)$.

bývá pro stromy typické, červeno-černý strom (dále RBT) má nejvýše logaritmickou časovou složitost pro všechny běžné operace (vkládání, hledání a mazání uzlu), neboli $O(\log n)$, kde n je počet uzlů stromu.

Vyvažování RBT je zajištěno pomocí těchto pravidel:

(1) Každý uzel je buď červený, nebo černý

(2) Každý list je černý

(3) Jestliže je uzel červený, pak má vždy 2 černé potomky

(4) Na cestě od kořene do libovolného listu stromu je stejný počet černých uzlů

Podívejme se blíže, na jejich význam.

Pravidlo (1) nám přidáním barevného atributu poskytuje analogii s 2-3-4 stromem, který užívá dvojuzly, trojuzly a čtyřuzly. Vzhledem k tomu, že 2-3-4 strom není předmětem této práce, bližší vysvětlení tohoto vztahu přenechávám k samostudiu.

Pravidlo (2) může nepoučenou osobu zaskočit ve významu, co se přesně myslí listem (listovým uzlem). U AVL stromu obsahoval každý list (terminální uzel) stejné množství informací jako kterýkoli neterminální uzel. U RBT stromu se však za list (terminální uzel) považuje až zarážka (*NULL*) označující konec větve. Zarážka dle tohoto pravidla si také nese svůj barevný atribut, ale nemá už žádné další významové položky. Protože RBT se tradičně kreslí *bez NULL uzlů* (implicitně se předpokládá jejich existence), tato drobná, ale podstatná změna pojmu chápání významu slova list nebývá patrná. Ukázka přesného zakreslení RBT se všemi listy je na 2.5 (str. 12). V dalších schématech však nebudou listy takto explicitně vyznačovány z důvodu již všeobecně zaběhnutého jednotného vykreslování všech stromů. V textu pojednávajícím o RBT se bude dále předpokládat význam slova list v pojetí RBT. Bude-li třeba zdůraznit význam tohoto označení, bude uvedeno označení jako „vnější“ list, list ve stejném významu jako v AVL stromu bude označen jako „vnitřní“ list.

Význam pravidla (3) tkví v tom, že ve stromu se zaručeně nemůžou vyskytovat libovolné dva červené uzly v bezprostředním přímém vztahu (otec – syn). Na druhou stranu je dobré si uvědomit, že toto platí jen pro červené uzly, nikoli pro uzly černé.

Poslední pravidlo (4) zajišťuje svým stejným počtem černých uzlů v *libovolné* cestě vyvažování černých uzlů. Platí, že *počet černých uzlů na cestě z kořene stromu do listu (mimo tento kořen) je dán počtem černých uzlů*[7]. V souvislosti s tímto pravidlem se často hovoří o tzv. černé výšce stromu. Ta je dána černou výškou kořene stromu[8]. Pro ukázku je tato černá výška stromu také vyznačena v 2.5 (str. 12). Zatímco pravidlo (3) svým omezením na červené uzly při realizaci vyvažování stromu způsobuje vznik černých uzlů, toto pravidlo zajišťuje vyvažování černých uzlů, a tak vyvážení celého stromu.

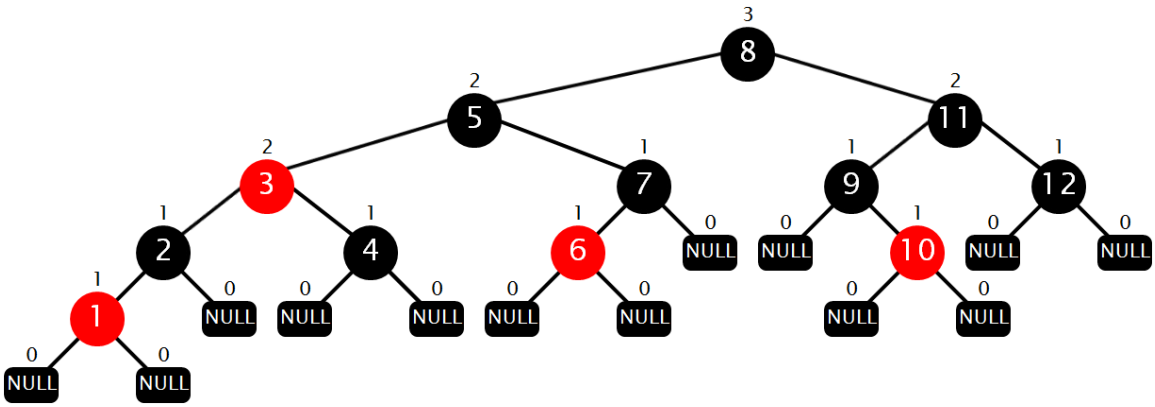
V různé literatuře (např. [24]) se někdy uvádí 5 vyvažovacích pravidel. Tím pravidlem navíc bývá, že kořen RBT je vždy černý. To je však především dáno pro snadnou implementaci úprav tohoto stromu. Důvod, proč je to právě barva černá souvisí s tím, že dle uvedených pravidel tato barva v kořenu stromu nemůže vůbec uškodit (žádné pravidlo nikdy neporuší).

Celkově uvedená pravidla zajišťují, že nejdelší cesta z kořene do listu není více jak dvakrát delší než nejkratší cesta z kořene do uzlu, neboli výška RBT s n vnitřními uzly je nejvýše

[7] Všimněme si, že správně při počítání černých uzlů se kořen nepočítá. Naopak se počítá vnější listový uzel.

[8] Černá výška stromu se počítá od (vnějších) listových uzlů, které mají vždy výšku 0. Každý uzel stromu má černou výšku danou počtem černých uzlů (z libovolné větve, tento počet je vždy stejný). V kořenu, který je černý (proč černý je řečeno dále v textu), se objevuje vždy hodnota černé výšky stromu.

$2 \cdot \log_2(n+1)$. Znamená to, že strom je přibližně vyvážený. Z implementačního hlediska tato pravidla zajišťují rotace (tak jako v jiných strukturách) a přebarvování uzlů. RBT má jen 2 základní jednoduché rotace. Jsou jimi *rotace doprava* a *rotace doleva*, jejich přesné pojmenování však není ustálené. Jednotlivé případy jejich efektu činnosti jsou uvedeny na shrnujícím obrázku 2.1 (případy 2.1(a) a 2.1(b), str. 5).



Obrázek 2.5: Červeno-černý strom (red-black tree)
Každý uzel je buď červený, nebo černý; každý list (*NULL*) je černý; každý červený uzel má 2 černé potomky; na cestě od kořene do libovolného listu stromu je vždy stejný počet černých uzlů. U neterminálních uzlů je vyznačena jejich černá výška, terminální (listové, *NULL*) uzly mají černou výšku 0.

Tak jako u AVL stromu, uspořádání uzlů se při vyhledávání nemění. Zato je nutné v případě vkládání či mazání uzlu kontrolovat platnost všech vlastností (pravidel) v RBT a při porušení zjednat jejich nápravu – provést rotace a přebarvování. I když je tento proces poměrně komplikovaný, má zachovanou svoji logaritmickou časovou náročnost.

Podrobnou analýzou RBT se zjistilo, že při *vložení* uzlu může dojít ke 3 odlišným případům, při kterých dochází k porušení pravidel RBT. Každý případ má i svůj zrcadlový protějšek. U *mazání* uzlu existují dokonce 4 různé případy, při kterých dochází k porušení RBT, každý případ má opět svůj zrcadlový protějšek. Projděme si jednotlivé situace a jejich principy vyvažování RBT s využitím nákresů v 2.6 (str. 14) pro vložení uzlu a 2.7 (str. 17) pro mazání uzlu. Pro jednoduchost jsou vždy zachyceny jen nejmenší počty směrodatných uzlů a jen případy pro vyrovnávání vzniklé v levé verzi stromu. Ty nastávají tehdy, byl-li přidaný uzel připojen od svého nastalého *prarodiče* vlevo nebo odebrán uzel od svého *rodiče* vlevo. Pravá verze vznikne pouze zrcadlovým obrácením jednotlivých případů, u kterých lze předpokládat, že si každý sám již odvodí.

Popis jednotlivých případů vyvažování RBT při vložení uzlu

Při vkládání nového uzlu do RBT platí zásada, že tento uzel má výchozí červenou barvu. Jinak by totiž žádný červený uzel nikdy nevznikl. Jeho připojení do RBT proběhne naprosto stejně jako v binárním vyhledávacím stromu. Pak je ale nutno strom po tomto vložení zkontrolovat a v případě porušení některého pravidla RBT opravit (přebarvit a provést rotace). Jaká pravidla můžou být vložením uzlu porušena? Pravidlo (1) nám určitě zůstane neporušeno, protože je dáno, že barva vloženého uzlu je červená. Pravidlo (2) zůstane neporušené také, protože vytváříme červený uzel s černými *NULL* potomky. Pravidlo (4) je vlastně pouhým vložením (napojením uzlu) také neporušeno, neboť vložený uzel sice nahradil někde ve

stromu černého *NULL* potomka, ale vkládaný uzel je červený a má také černé terminální potomky, takže počet uzlů v libovolné cestě se nezmění. Protože se vkládá červený uzel, může dojít jen k porušení pravidla (3) tehdy, pokud rodič vloženého uzlu bude také červený. Proto se při vložení uzlu u vyvažování RBT snaží přesunout tato anomálie směrem ke kořenu RBT při zachování platnosti všech (neporušených) pravidel, neboli zachovat je jako invarianty při vyvažování stromu. Pro ukončení vyvažování existují jen jediné 2 možnosti. Buď se provedla taková rotace či přebarvení, že při ní došlo k odstranění přímé sousednosti červených uzlů, nebo se došlo do kořene stromu, který se závěrem přebarví vždy černě, čímž se vyvaruje 2 červeným uzlům mezi kořenem a jeho některým potomkem. Jak lze tušit, největší pozornost v jednotlivých případech je třeba věnovat zachování invariantu pravidla (4).

Případ 1 (viz 2.6(a), str. 14): Tento případ se zásadně odlišuje od následujících situací tím, že strýc/teta (označen jako Y) vloženého uzlu X je červený. Jak si lze všimnout, nezáleží na tom, zda vložený uzel byl připojen ke svému rodiči zleva nebo zprava, barevné označení rodiče, prarodiče a strýce je jednoznačně stanoveno[9]. Všimněme si, že tento případ provede jen přebarvení všech uzlů, přičemž prarodič je přebarven červeně bez znalosti, zda rodič tohoto prarodiče (X') není náhodou červený. Tento případ tedy provádí pouhé přesunutí problému blíže ke kořenu stromu. Počty černých uzlů ve všech cestách zůstávají zachovány.

Případ 2 (viz 2.6(b), str. 14): Tento případ provádí pouze transformaci (jednoduchou rotaci) kolem rodiče vloženého uzlu, čímž se vygeneruje případ 3. Není na něm nic víc zvláštního. Protože pracuje jen s červenými uzly, počet černých uzlů v libovolné cestě se zákonitě nezmění.

Případ 3 (viz 2.6(c), str. 14): Tento případ provádí rotaci kolem prarodiče vloženého uzlu. Tato rotace je opačného smyslu než v předchozím případě, jedná-li se o stejnou verzi vyvažování. Aby se zachovala platnost pravidla (4), je nutné navzájem přehodit barvy rodiče a prarodiče vloženého uzlu. Zároveň se tím docílí, že pozice černé barvy zůstane zachována, takže nemůže dojít k přesunutí problému porušení pravidla (3) jako v prvním případu. Vyvažování stromu zcela jistě na případu 3 skončí.

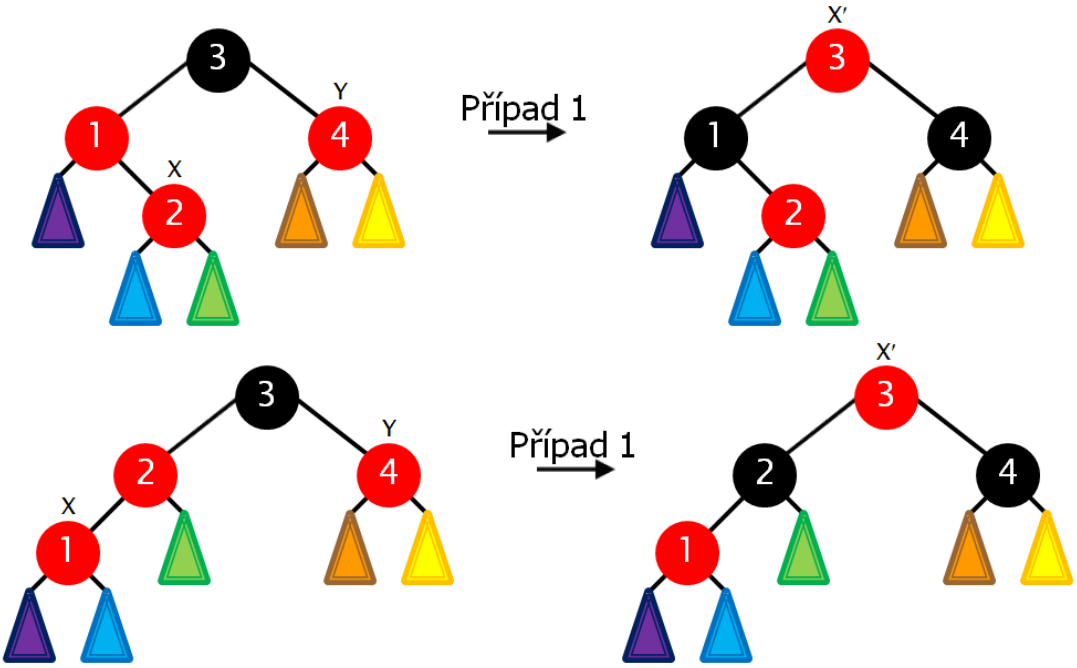
Nyní se již můžeme vyjádřit k maximálnímu počtu rotací, které mohou nastat při vložení uzlu do RBT. Jedná se nejvýše o 2 rotace a to právě z případů 2 a 3. Případ 1 provádí jen přebarvování uzlů, žádnou rotaci nevolá.

V souvislosti s případy 2 a 3 lze ještě upozornit na jednu zajímavou drobnost. Pokud srovnáme počáteční stav situace z případu 2 a následně koncový stav z případu 3, shledáme, že tento stav je co do umístění uzlů (v tomto případě levých verzí) naprosto stejný jako rotace DLR v AVL stromu. Je to ukázka na ověření tvrzení, že všechny dvojité rotace jsou odvozeny z jednoduchých rotací, pokud jsou provedeny nad správným uzlem a ve správném pořadí.

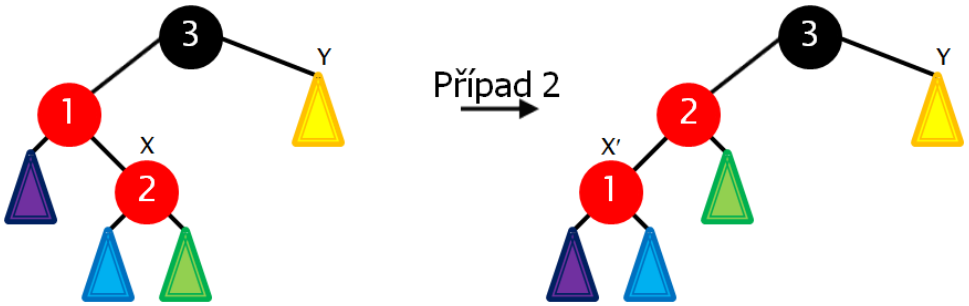
Popis jednotlivých případů vyvažování RBT při smazání uzlu

Smazání (odstranění) libovolného uzlu z RBT je vůči vložení uzlu o něco komplikovanější, což není nic překvapivého. Při úvahách (a následně i implementaci) u mazání uzlu se využívá

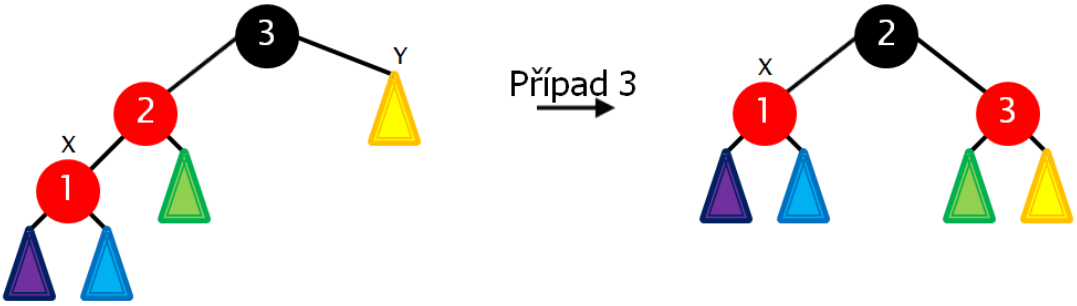
[9] Barva rodiče je červená proto, protože by jinak nevznikl žádný problém při vložení uzlu, barva strýce je červená, protože je tím tento případ charakteristický, a barva prarodiče je nutně černá proto, protože před vložením uzlu to byl platný RBT – nemůže být rodič a prarodič červené barvy



(a) Případ 1



(b) Případ 2



(c) Případ 3

Obrázek 2.6: Možné případy uspořádání uzlů v RBT stromu po *vložení uzlu značeném* X vedoucí k nutnosti provést vyvážení (Y značí strýce uzlu X, označení s čárkou ($'$) po provedení daného případu znamená posun na vyznačený uzel do další iterace vyvažování RBT)

toho, že vnější list (*NULL*), který je vždy černý, se dočasně považuje za plnohodnotného potomka jistého uzlu a tento uzel za rodiče uzlu *NULL*. Základem rušení uzlu v RBT je naprosto stejné provedení jako v binárním vyhledávacím stromu, neboli ruší se fyzicky ten uzel, který má nejvýše jednoho potomka. Po zrušení uzlu je pak nutné zkontrolovat, zda nedošlo k porušení vlastností RBT. K tomu dojde, pokud rušený uzel (označme si ho např. Y) je právě černý, protože některá cesta ve stromu bude obsahovat o jeden černý uzel méně, čímž se poruší pravidlo (4). Při vyrovnávání odstraňování se tedy snažíme do této cesty dodat černou barvu. Nejjednodušším řešením by bylo přesunout černou barvu z odstraněného uzlu Y na jeho (jediného) potomka X, který může být neterminálním uzlem, ale také i listem (*NULL*). Pokud je potomek červený, stačí jen provést jeho přebarvení. Problém však nastane, když tento potomek X je černý a měl by mít najednou 2 barvy černé, čímž by se porušila vlastnost pravidla (1). Při vyvažování se tedy snažíme obnovit platnost pravidla (1) tím, že posunujeme černou barvu postupně ke kořenu stromu dokud uzel X (s černou barvou navíc získanou od odstraněného uzlu Y) není červený (přebarvil by se na černo), neprovede se nějaká vhodná rotace a přebarvení uzlů umožňující se „zbavit“ černé barvy navíc nebo není kořenem („ztráta“ černé barvy uzlu nad kořenem nevadí) (interpretace dle [3]). Jiný a možná jednodušší pohled na interpretaci nastalé situace než přesouvání černé barvy je ten, že se snažíme do cesty s aktuálně chybějící černou barvou odstraněného uzlu Y umístit takový uzel, kterému by šlo černou barvu dodat. Pak se bude jednat přímo jen o dodržení platnosti pravidla (4), i když v obou pohledech je vždy patrná klíčová myšlenka zachování počtu černých uzlů na cestě z rodiče do listu.

Než přikročíme na jednotlivé situace (viz 2.7, str. 17), zdůvodněme si existenci, situaci a barvu značených uzlů X a W ve všech případech. Na základě uvedených podmínek ukončení vyvažování platí, že uzel X je nutně černý. Také v jeho cestě chybí černá barva, neboli uzel X by měl mít dvě černé barvy. Dále platí, že uzel X není kořenem RBT, ale jak už bylo řečeno, může být vnějším listem (*NULL*). Protože X není kořenem, musí existovat jeho sourozenec W, který může mít libovolnou barvu. Tento sourozenec však určitě nemůže býti vnějším listem, protože v cestě přes uzel X je o jeden černý uzel méně. Lze tedy ve všech případech počítat s tím, že uzel W má své nějaké dva potomky (za potomka může mít i vnější list). Jednotlivé případy, které je třeba ošetřit, tedy mají naprosto stejné uspořádání známých uzlů vzájemně se lišící tím, že uzly, o kterých víme, že určitě existují, mají různé barvy.

Případ 1 (viz 2.7(a), str. 17): Tento případ je charakteristický tím, že sourozenec uzlu X (značíme ho W) má právě červenou barvu. Společný rodič uzlu X a W musí být černý, protože by to jinak už i před smazáním uzlu Y nebyl RBT strom – měl by porušené pravidlo (3). Díky červené barvě W však musí být i oba potomci uzlu W černí. Díky tomu můžeme zaměnit barvu W s jeho rodičem a provést rotaci okolo tohoto rodiče (rodič X, W) bez jakéhokoli ovlivnění situace v RBT. Význam případu 1 tedy spočívá v transformaci této situace na jeden z případů 2, 3 nebo 4.

Případ 2 (viz 2.7(b), str. 17): Tento případ má černé oba potomky uzlu W a zároveň černý uzel W. Lze tedy vzít černou barvu z W a posunout ji do rodiče W, čímž dodáme černou barvu do cesty přes uzel X. W obarvíme červeně. Zajistíme tak, že počet černých uzlů přes uzel W se nezmění. Vše by bylo moc pěkné, kdyby bylo zaručeno, že rodič uzlu X a W je červený a může se mu tak tato černá barva předat. Vyvažování by tak došlo ke konci. Bohužel to je zaručeno jen tehdy, narazíme-li na tento případ přes případ 1. Není-li jejich společný rodič červený, došlo jen k posunu problému o uzel výše ke kořenu stromu,

což je význam tohoto případu. Všimněme si, že tento případ provádí jen přebarvování uzlů, žádné rotace.

Případ 3 (viz 2.7(c), str. 17): Tento případ nastane, pokud uzel W je černý a jeho levý potomek (v probírané levé verzi) je červený a pravý černý. Bez narušení vlastností RBT lze přehodit barvy mezi W a jeho levým potomkem a provést rotaci kolem W. Nový sourozenec uzlu X (W') zůstal černý, ale má již červeného pravého potomka. Význam případu 3 je tedy v transformaci této situace na poslední případ 4.

Případ 4 (viz 2.7(d), str. 17): Tento případ má černého sourozence W a (v probírané levé verzi) červeného pravého potomka uzlu W. Levý potomek uzlu W může mít libovolnou barvu. Záměnou barvy uzlu W a jeho rodiče (o libovolné barvě) a provedením rotace kolem rodiče uzlu W způsobíme, že dodáme do cesty přes uzel X černý uzel, přičemž barva kořenového uzlu tohoto případu zůstane zachována. Provedením rotace bychom však zároveň odebrali černý uzel z pravé větve rodiče uzlu W. Proto nesmíme zapomenout navíc přebarvit pravého potomka uzlu W černě, aby po provedení rotace byly nadále splněny všechny vlastnosti RBT. Takovéto přebarvení si zde můžeme beztrestně dovolit, protože máme v tomto případu zaručeno, že pravý potomek uzlu W je na počátku červený. Význam případu 4 je ten, že se zcela jistě po jeho provedení naplní všechny požadavky na splnění všech pravidel RBT, čímž vyvažování skončí.

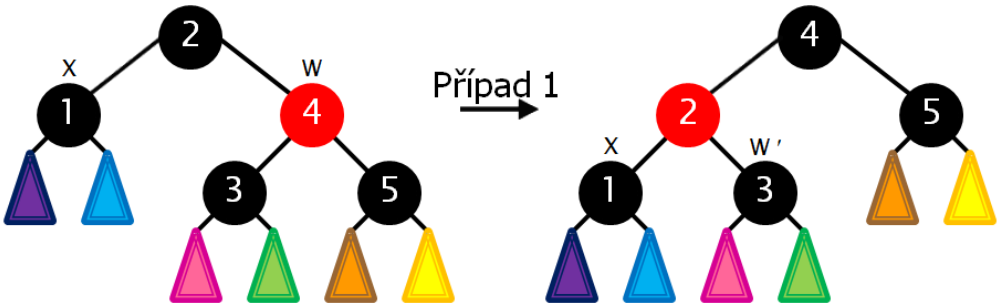
Vyvažování RBT lze ukončit jedině v případu 2 a 4, přičemž v případu 4 tento konec nastává nepodmíněně. Nejvyšší počet rotací je 3 a to tehdy, když nastane případ 1, který se ztranformuje na případ 3 a ten na případ 4. V případu 2 se provádí jen přebarvování a posun o uzel blíže ke kořenu stromu v čase nejvýše $O(\log n)$. Proces vyvažování tedy pracuje v čase $O(\log n)$ s provedením nejvýše 3 rotací.

Typickým uplatněním červeno-černého stromu (red-black tree) jsou samovyvažující asociativní pole (např. funkcionální programování), hodí se pro organizování jednotek vzájemně porovnávatelných dat. Vůči rozvinutému stromu (splay tree) či přeskakujícímu seznamu (skip list) je RBT složitý, ale jeho operace (jako je vkládání, hledání, mazání) trvají nejvýše $O(\log n)$ času, kde n je počet uzlů ve stromu. Proto mají uplatnění v aplikacích, které tyto operace potřebují zrychlit. Ačkoli by se mohlo zdát, že tato struktura je při existenci AVL stromu zbytečná, měla by mít oproti AVL stromu zrychlené vkládání a hlavně mazání dat, protože nemá tak přísné požadavky na vyváženost uzlů. Například odstranění jednoho uzlu z AVL stromu o n uzlech může potřebovat stejný počet rotací jako je počet uzlů na cestě ke smazanému uzlu. Tatáž operace v červeno-černém stromu nikdy nepřesáhne více než 3 rotace. Je nutno si však uvědomit, že zákonitě dochází k mírnému zpomalení vyhledání konkrétního uzlu vůči AVL stromu. Proto lze považovat červeno-černý strom vhodnější pro časté vkládání a odstraňování dat (uložených v uzlech, pro „průvan dat“), než-li AVL strom, který je více vyvážený a zaměřený tak na vyhledávání. Za zajímavou vlastnost RBT lze považovat, že při vhodné implementaci umožňuje mít klíče neunikátní, přičemž všechny operace umožní zachovat stabilitu (pořadí uzlů se stejným klíčem).

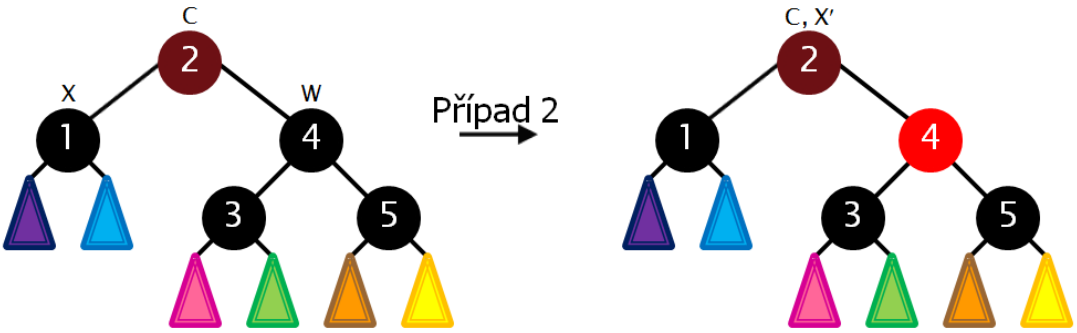
2.1.3 Rozvinutý strom (splay tree)

Rozvinutý strom (dále SPT)[10] je prostý binární vyhledávací strom se schopností přizpůsobování se požadavkům. Jinými slovy, strom má vlastnost takovou, že prvky, ke kterým se

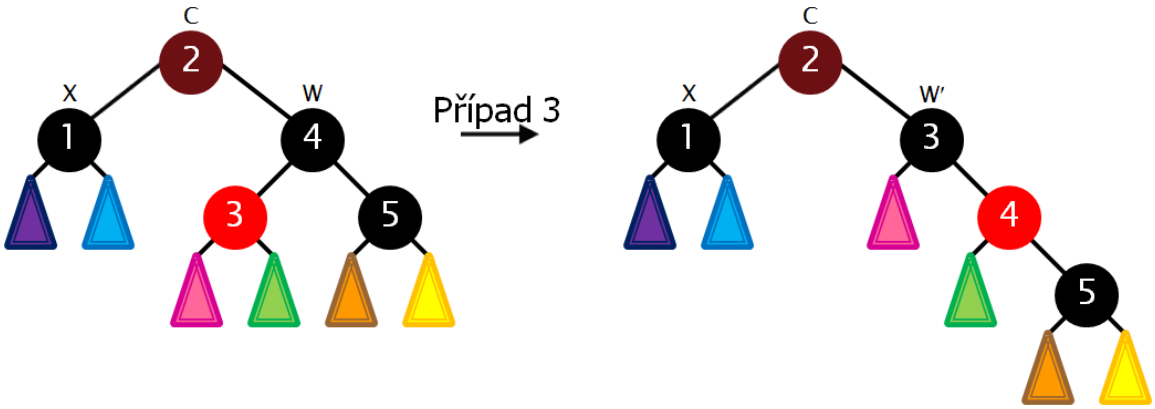
[10] Při zpracování bylo čerpáno z těchto studijních materiálů: [4, 7. monografie], [14] a [26]



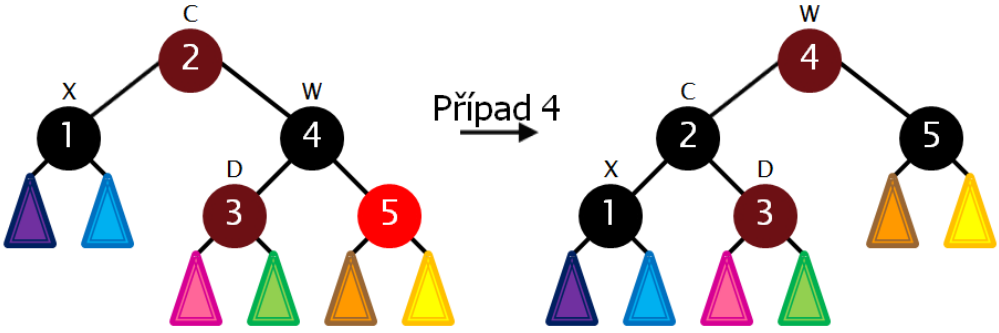
(a) Případ 1



(b) Případ 2



(c) Případ 3



(d) Případ 4

Obrázek 2.7: Možné případy uspořádání uzlů v RBT stromu po *odstranění uzlu* Y, jehož pozůstalý přímý potomek je značen X, vedoucí k nutnosti provést vyvážení (W značí sourozence uzlu X, C a D značí uzel o libovolné barvě, označení s čárkou ($'$) po provedení daného případu znamená použití tohoto uzlu do další iterace vyvažování RBT)

nedávno přistupovalo, jsou znovu rychle dostupné. Protože se jedná jen o takto rozšířený binární vyhledávací strom (dále BVS), může nastat zdegenerování stromu a přiblížení se tak až k lineární časové složitosti. Naštěstí tento stav netrvá dlouho a platí, že amortizovaná časová složitost je logaritmická. Tento strom byl vynalezen Danielem Sleatorem a Robertem Tarjanem v roce 1985.

Práce se stromem je založena na splay operaci (splay = rozšířit, rozvinout). Ta modifikuje strom tak, aby se tázaný uzel při respektování uspořádanosti vyhledávacího stromu dostal do kořene SPT. Pokud se má provést nějaká činnost se stromem (jako je vložení, vyhledání nebo smazání uzlu), je nejdříve provedena splay operace. Po ní se pak provede vlastní žádaná činnost. Je evidentní, že provedení kterékoli dále požadované činnosti nad SPT je od BVS výrazně jednodušší, protože stačí v těchto případech porovnat jen zpracovávaný klíč uzlu s hodnotou klíče kořene SPT. Vše ale závisí na splay operaci. Způsob její činnosti je založen na dvou krocích, jimiž jsou (1) vyhledání žádaného uzlu a (2) provedení série vhodných rotací tak, aby se uzel dostal do kořene stromu.

Bottom-up splay

Vyhledávání uzlu v rámci splay operace je stejné jako vyhledávání v BVS. Není-li hledaný uzel dostupný (ve struktuře neexistuje), bude se přesouvat nejbližší příbuzný hledaného uzlu (předpokládaný přímý rodič nenalezeného uzlu). K přesunům slouží celkem 6 rotací, které lze rozdělit na levou a pravou verzi do 3 odlišných případů. Pokud se jedná o levou rotaci (rotace zleva), označujeme tuto rotaci jako *zig*. Je-li to rotace pravá (zprava), jedná se o rotaci označovanou jako *zag*. *Zig* a *zag* rotace tvoří jednoduché rotace. Jejich vzájemným složením dostaneme 4 dvojité rotace, které nazýváme *zig-zag*, *zag-zig*, *zig-zig* a *zag-zag*. Jejich projev účinků můžeme pozorovat na obrázku 2.1 (str. 5). Uplatňování jednotlivých rotací na umístění přesouvaného uzlu závisí na uspořádání stromu. Rotace jsou vybírány a aplikovány do té doby, než se vyskytne přesouvaný uzel v kořenu SPT.

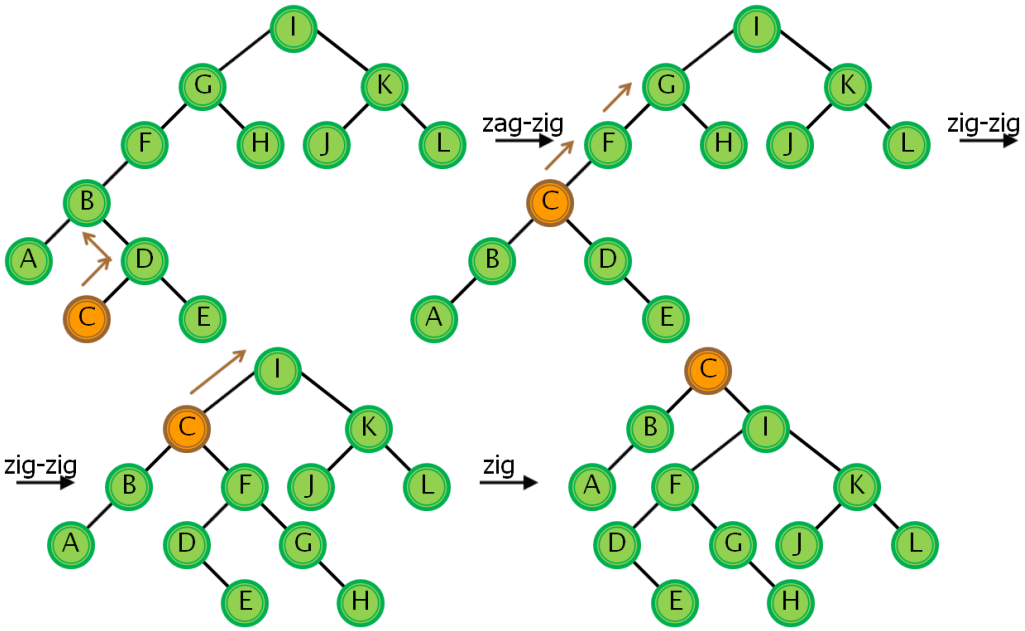
Označme přesouvaný uzel jako x, rodiče (parent) tohoto uzlu jako $p[x]$ a prarodiče jako $p[p[x]]$. Pokud x nemá svého rodiče $p[x]$ (takový uzel neexistuje), přesouvaný uzel x je už na požadovaném místě – v kořenu stromu. Pokud x nemá svého prarodiče $p[p[x]]$, tak se aplikuje jednoduchá rotace, která závisí jen na tom, zda uzel x je levý nebo pravý potomek svého rodiče $p[x]$. Pokud má x svého prarodiče $p[p[x]]$, aplikují se dvojité rotace, jejichž výběr závisí na tom, zda uzel x je levý nebo pravý potomek svého rodiče $p[x]$ a zda uzel $p[x]$ je levý nebo pravý potomek svého rodiče $p[p[x]]$. Dokud uzel není v kořenu stromu, provádí se uvedené porovnávání. Volbu vhodné rotace přehledně znázorňuje následující tabulka.

	$p[x] < p[p[x]]$	$p[x] > p[p[x]]$	not exist $p[p[x]]$
$x < p[x]$	zig-zig	zag-zig	zig
$x > p[x]$	zig-zag	zag-zag	zag

Pro přehledné znázornění a uvědomění si, jak bottom-up splay operace pracuje, je jako ukázka její demonstrace činnosti uvedena na obrázku 2.8 (str. 19). Z něj je dobře patrné, jak požadovaný uzel je ze své aktuální pozice neustále rotacemi posunován až do kořene SPT.

Top-down splay

Popsaný způsob bottom-up splay operace je sice pěkný, ale zajisté jste si všimli, že je potřeba nejdříve vyhledat uzel a pak jej přemístit do kořene stromu. Je tak nutné v podstatě procházet strom dvakrát, což může být neefektivní. Proto byla navržena top-down splay operace, která tyto 2 kroky vzájemně spojuje.

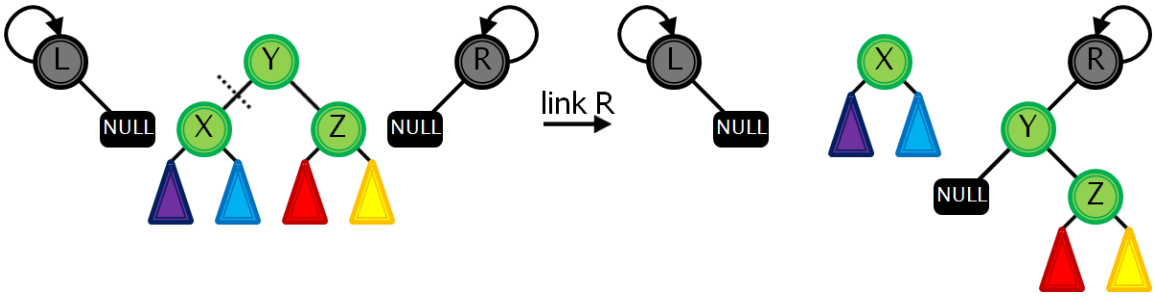


Obrázek 2.8: Ukázka postupu modifikace stromu při aplikaci *bottom-up splay* na vyznačený uzel C

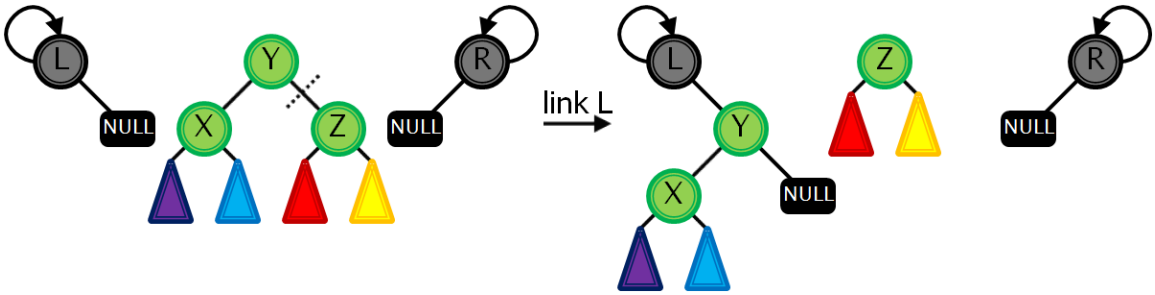
Strategie top-down splay operace je založena na tom, že rozděluje procházený strom do 3 stromů – levý, střední (někdy označovaný jako prostřední) a pravý. Střední strom obsahuje tu část SPT, která ještě nebyla projita. Jedině v něm se může nacházet hledaný uzel. Levý strom postupně obsahuje ty uzly, o kterých se ví, že jsou menší než kterýkoli uzel prostředního stromu a pravý strom postupně obsahuje ty uzly, které jsou rozpoznány býti většími než kterýkoli uzel prostředního stromu (rozhoduje se dle porovnání klíčů uzlů). Na začátku je levý a pravý strom prázdný, všechny uzly SPT jsou ve středním stromu. Když se začne procházet od kořene k žádanému uzlu, vykonávají se potřebné splay rotace a rozdělování různých větví středního stromu (shluků uzlů) s připojováním do levého či pravého stromu. Když je dosaženo konce procházení prostředního stromu, protože žádaný uzel byl již v něm nalezen nebo se ve stromu nenacházel, dojde k sestavení všech 3 stromů do jediného tak, že levý strom se připojí k aktuální části prostředního stromu vlevo a pravý vpravo, čímž se dosáhne toho, že požadovaný uzel (nebo jeho přímý předpokládaný rodič, pokud žádaný uzel ve stromu vůbec neexistoval) se objeví v kořenu SPT.

Top-down splay operace narozdíl od bottom-up splay má jiné požadavky na úpravy SPT. Top-down splay užívá *připojení vpravo* (*link right*), *připojení vlevo* (*link left*), *rotaci doprava* (*rotate right*), *rotaci doleva* (*rotate left*) a *sestavení* (*assemble*). Jak plyne z názvů jednotlivých úprav, připojení provádí oddělení jistých uzlů ze středního stromu a připojení jich do stromu na stranu dle příslušného typu připojení, rotace provádí naprosto stejné úpravy prostředního stromu jako v předchozím RBT a sestavení provádí spojení 3 stromů do jediného stromu tak, aby bylo zachováno uspořádání uzlů a kořen prostředního stromu se stal kořenem v SPT. Schématické znázornění vyjmenovaných úprav je na obrázku 2.9 (str. 20).

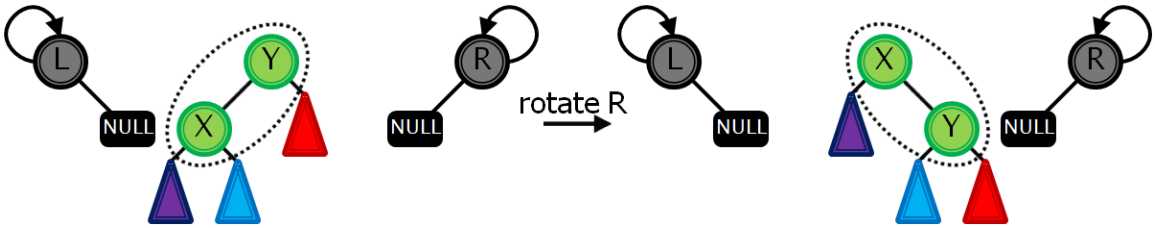
Označíme-li aktuální kořenový uzel jako x, jeho (přímého) potomka (descendant) na cestě k hledanému cíli jako $d[x]$ a jeho prapotomka k hledanému cíli jako $d[d[x]]$, můžeme snadno popsat, kdy provádět které uvedené úpravy. Pokud x je žádaný uzel nebo x nemá



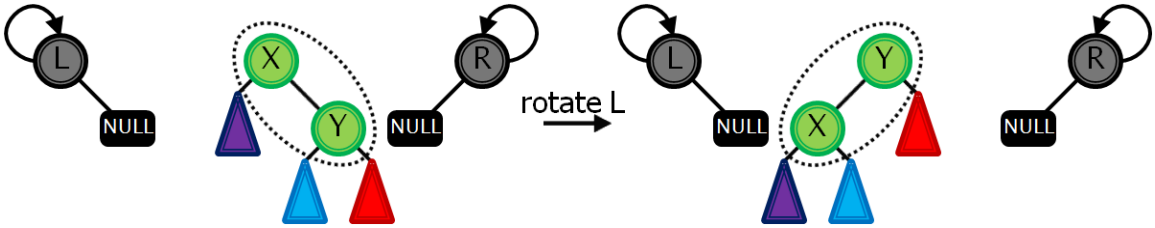
(a) Připojení vpravo (link right)



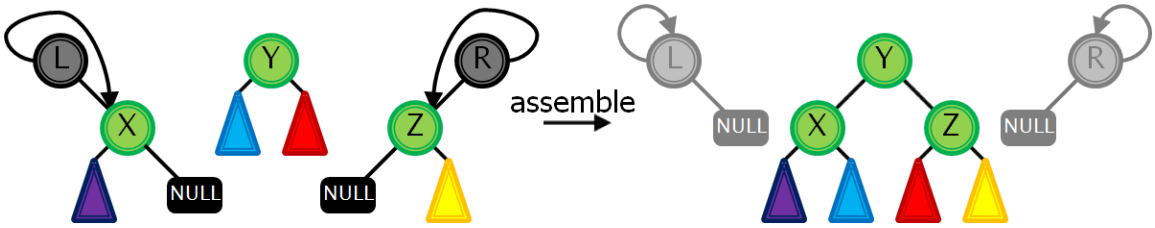
(b) Připojení vlevo (link left)



(c) Rotace vpravo (rotate right)



(d) Rotace vlevo (rotate left)



(e) Sestavení (assemble)

Obrázek 2.9: Znázornění činnosti jednotlivých úprav užívaných v *top-down splay*

svého potomka $d[x]$ ($d[x]$ uzel neexistuje), na pozici kořene je už správný uzel, nelze dostat lepší. Pokud x nemá svého prapotomka $d[d[x]]$, tak se aplikuje připojení, které závisí jen na tom, zda uzel x (rodič) má svého potomka $d[x]$ vpravo nebo vlevo. Pokud existuje prapotomek $d[d[x]]$ k uzlu x ($d[d[x]]$ leží na cestě k žádanému uzlu), aplikují se buď dvě různá připojení nebo rotace a připojení (prvně rotace, pak připojení). Jejich výběr závisí na tom, zda uzel x je pravým nebo levým rodičem svého potomka $d[x]$ a zda potomek $d[x]$ je pravý nebo levý rodič svému potomkovi $d[d[x]]$. Správný výběr vhodné úpravy přehledně znázorňuje následující tabulka.

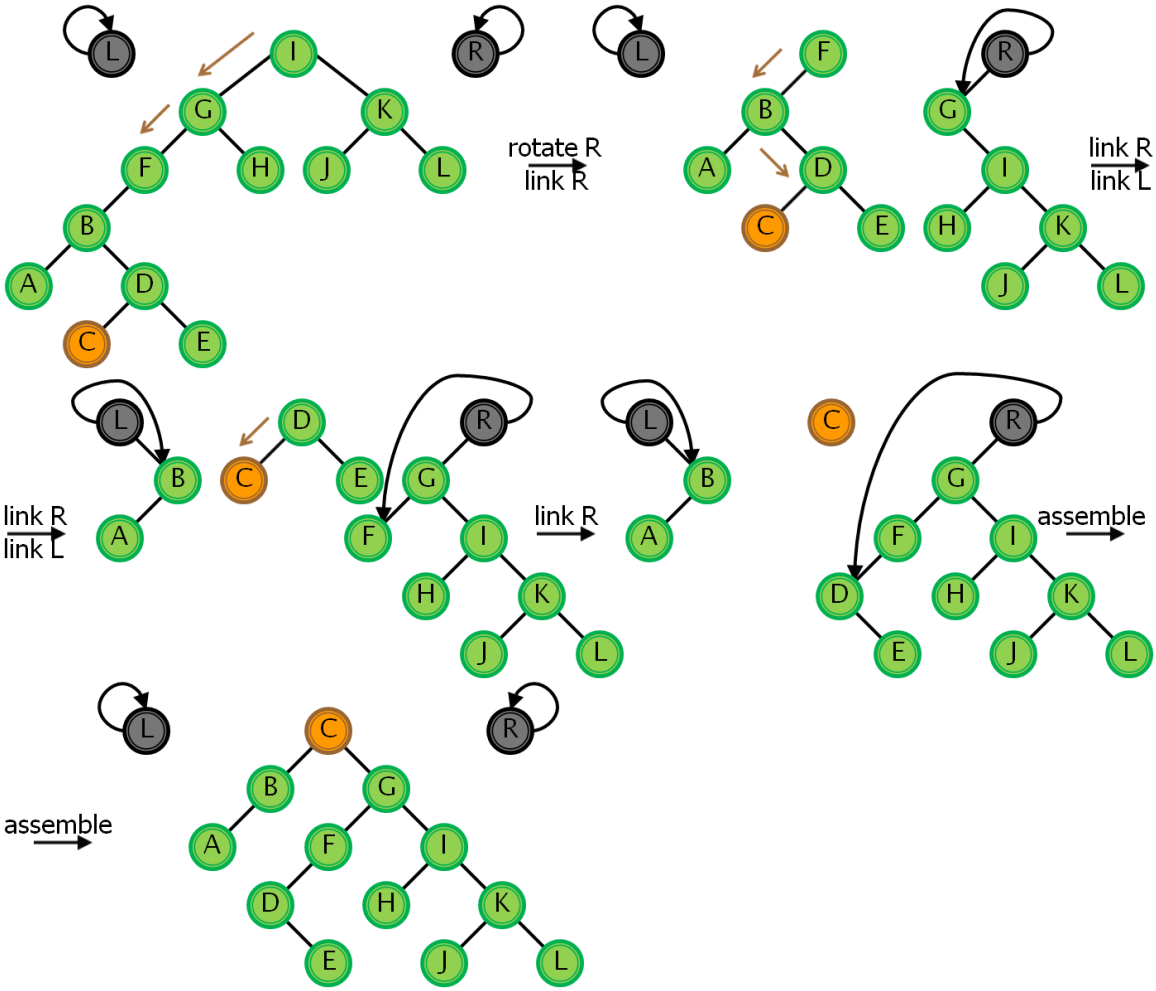
	$d[d[x]] < d[x]$	$d[d[x]] > d[x]$	not exist $d[d[x]]$
$d[x] < x$	rotate right, link right	link right, link left	link right
$d[x] > x$	link left, link right	rotate left, link left	link left

Pokud je již žádaný uzel v kořenu středního stromu nebo nelze jej najít, provede se sestavení všech 3 stromů do jediného, čímž operace top-down splay definitivně končí.

Pro přehledné znázornění top-down splay operace je jako ukázka její demonstrace přiložen obrázek 2.10 (str. 22). Na rozdíl od předchozího způsobu, strom je měněn nikoli přesouváním žádaného uzlu, ale postupným odřezáváním kořene k hledanému uzlu. Levý strom je označen L (left), pravý R (right). Kořenové uzly L a R těchto stromů, které pokládáme za jejich hlavičky, nenesou žádná data, jsou využity jen jejich levé a pravé ukazatele s mírně pozměněným významem jejich použití. *V levém stromu* jsou všechny uzly připojovány vždy k nejpravějšímu uzlu dosud existující části vytvářeného stromu, aby se zachovalo pravidlo uspořádanosti uzlů. Je-li strom prázdný, dojde k připojení uzlů k pravému ukazateli hlavičky levého stromu. Protože zjišťovat nejpravější uzel levého stromu pro každé připojení dalších uzlů by tak bylo málo efektivní, je využit levý ukazatel, který je po vložení uzlů vždy nastaven na uzel, ke kterému příště připojit (vpravo) další uzly z prostředního stromu. Shrňme si to. Pravý ukazatel hlavičky nám neustále ukazuje na neměnný kořen levého stromu, zatímco levý ukazatel ukazuje na uzel, ke kterému příště vpravo připojit obdržené uzly (větve prostředního stromu). *V pravém stromu* je využití ukazatelů analogické s tím, že levý ukazatel hlavičky ukazuje na neměnný kořen pravého stromu, zatímco pravý ukazatel ukazuje na uzel, ke kterému příště (vždy) vlevo připojit obdržené uzly (větve prostředního stromu). Na začátku, kdy je levý a pravý strom prázdný, jsou ukazatelé na kořen příslušného stromu nastaveny na *NULL*, zatímco druhý ukazatel je nastaven sám na sebe (právě k hlavičce se bude příště nějaký uzel připojovat).

Po krátkém zamyšlení můžeme říci, že jednoduché případy (jednoduchá rotace: zig, zag, případ jediného připojení: vlevo, vpravo) nastávají v jedné splay úpravě stromu nejvýše jednou, zatímco složené případy (dvojité rotace, více připojení či kombinace rotace a připojení) můžou být volány tolikrát, kolik je poloviční počet uzlů na cestě mezi kořenem SPT a žádaným uzlem umisťovaným do tohoto kořene. Pro rychlé provedení splay operace je tedy důležité, aby žádané uzly nebyly příliš vzdálené od kořene, neboli byl užíván jistý nepříliš rozsáhlý počet uzlů výrazně častěji než ostatní uzly.

Díky tomu, že je strom adaptibilní, neboli že se sám optimalizuje pro přístup k naposledy užívaným uzlům, lze očekávat jisté zlepšení výkonnosti u často vyhledávaných uzlů (jejich klíčů), které tak budou k dispozici dříve. To lze považovat za užitečné do praxe. Princip adaptibility však není nic nového, obdobnou přizpůsobivost nalezneme např. v lineárních strukturách, kde se každý vyhledaný uzel prohodí se svým předchůdcem. Z hlediska implementace se bude tvořit snadněji než jiné (především samovyvažující) stromy (AVL strom, červeno-černý strom). Nepotřebuje si pamatovat další pomocné informace, a proto má nižší paměťové nároky (stejné jako BVS). Všechny operace zachovávají pořadí identických klíčů, což je jistě zajímavá vlastnost umožňující mít i nejednoznačné klíče (většinou se jí vyznačují



Obrázek 2.10: Ukázka postupu modifikace stromu při aplikaci *top-down splay* na vyznačený uzel C
Ačkoli se jedná o stejný počáteční strom jako v 2.8 a do kořene se přesouvá stejný uzel, výsledný strom po provedení top-down splay operace nemusí být nutně stejný jako při použití bottom-up splay.

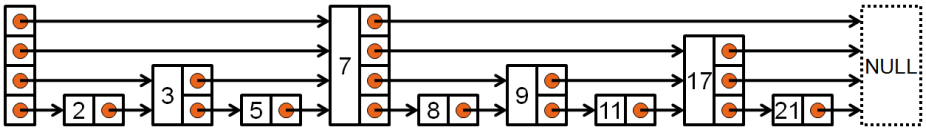
jen stabilní třídící algoritmy nad lineárními strukturami). Dobře provedená operace např. vyhledávání umožňuje vrátit uzel nejvíce nalevo či napravo. Na druhou stranu, struktura se neustále mění s každým dotazem včetně vyhledávání. (Ostatní stromy se při vyhledávání nemění.) Pokud se bude přistupovat ke všem prvkům stromu se stejnou pravděpodobností, výkon stromu bude podstatně horší, a to i vůči jednoduchému nijak nevyvažovanému binárnímu stromu. Navíc nelze u tohoto stromu zaručit, že bude mít logaritmickou časovou složitost, protože může teoreticky zdegenerovat až na lineární seznam obdobně jako BVS. Proto např. nelze tuto vyhledávací metodu použít v časově kritických aplikacích.

2.1.4 Přeskakující seznam (skip list)

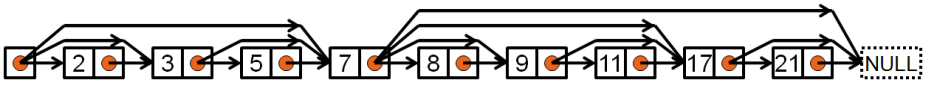
Přeskakující seznam[11] je datová struktura vyhledávací metody, která ve svém základu si uchovává paralelně uspořádané lineární seznamy, které jsou vzájemně propojené. Lze říci, že je to pravděpodobnostní alternativa vyvažovaných stromů. Její efektivita je údajně srovnatelná s binárním vyhledávacím stromem. Přeskakující seznam (dále SKL) byl navržen roku 1987 Williamem Pughem, ale byl popsán až v červnu 1990, což je považováno za datum vzniku této vyhledávací metody.

Při hledání v přeskakujícím seznamu se rychle přechází (přeskakují) irelevantní části seznamu (odtud skip list). Proto vyhledávání, vkládání a mazání je možné s logaritmickou složitostí. Užívaný seznam je vždy uspořádaný[12]. Prvky (položky, uzly) tohoto seznamu jsou vzájemně propojené ukazateli na svého následníka, přičemž každý prvek může mít těchto ukazatelů několik. Vždy však má alespoň jeden na svého přímého následníka. Pro vysvětlení, pochopení a znázorňování SKL se užívá úrovňová (vrstvová) reprezentace. Každá vyšší vrstva seznamu vždy tvoří tzv. *rychlou linku* (*express lane*) vrstvě nižší. Uzel, který má dopředné (přeskakující) ukazatele, označujeme jako úrovňový uzel (level node).

V dnešní době existují dvě verze SKL. Jsou jimi *klasický přeskakující seznam* (*classic skip list*) a *spojovaný přeskakující seznam* (*linked skip list*). Jejich odlišnost tkví jen v realizaci dopředných ukazatelů. Ta je přehledně znázorněna v 2.11. V klasickém SKL má každý uzel své pole ukazatelů o známé velikosti. Klasický SKL tak užívá nastavovaný seznam polí odkazů na další uzly. Spojovaný SKL místo toho užívá jen dynamicky vytvářenou dvojrozměrnou síť odkazů. Má tak díky tomu menší paměťové nároky, ale je více implementačně náročný, má obtížnější a složitější implementaci především pro vkládání a mazání. Lze očekávat, že se to negativně projeví právě na rychlosti těchto operací. Navíc pro poměrně nízký počet úrovní, který je i pro rozsáhlé množství uzlů potřeba, je úspora paměťového prostoru (zvlášť v dnešní době) stejně nevalná. Z těchto důvodů je logické, že se tato složitější varianta příliš v praxi nepoužívá a nebývá mnohdy vůbec uváděna. Proto jsem se v následné implementaci SKL přidržel její klasické podoby dle jejího původního návrhu. Ze stejného důvodu bude i další popis orientován právě na tuto verzi.



(a) Klasický přeskakující seznam (classic skip list)



(b) Spojovaný přeskakující seznam (linked skip list)

Obrázek 2.11: Znázornění rozdílu mezi *klasickým přeskakujícím seznamem* a *spojovaným přeskakujícím seznamem*

Nejdůležitější činností v SKL je tvorba jednotlivých úrovní seřazených seznamů. Děje se tak povýšením některého uzlu do vyšší úrovně, typicky při vkládání uzlu (viz dále). Každý uzel má tolik ukazatelů, v kolikáté je úrovni. Všechny ukazatele jsou vždy nastaveny tak, aby ukazovaly na prvek svého přímého následníka v dané úrovni. Vyjádření úrovně, na kterou

[11] Při zpracování bylo čerpáno z těchto studijních materiálů: [12], [17] a [25]

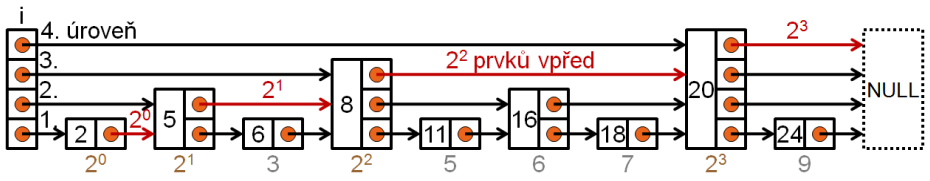
[12] Užívá se vzestupného uspořádání, i když principiálně opačné uspořádání nevadí, záleží na implementaci.

bude prvek povýšen, je dáno pravděpodobností s binomickým rozdělením (rozložením). Toto rozdělení je známé tím, že výskyt náhodného jevu (při n nezávislých pokusech) má stále stejnou pravděpodobnost p. Toto p se před použitím SKL volí, nejčastěji to bývá $p = 1/2$ či $p = 1/4$, protože jsou při něm obdrženy nejrychleji relevantní výsledky. Lze říci, že při snižování pevně dané pravděpodobnosti p se logaritmická časová složitost operace ztrácí a výsledky se přibližují ke složitosti prostého jednosměrně vázaného seznamu (tedy k lineární časové složitosti). Důvodem ke snižování p bývá potřeba omezit počet úrovní a snížit tak paměťovou náročnost, i když množství prvků, které bude SKL obsahovat, bude ponecháno. Další text bude předpokládat, že je zvoleno $p = 1/2$, není-li uvedeno jinak. Taktéž v části o implementaci SKL (v 2.2.4, str. 37) bude pracováno s touto hodnotou, protože tvoří stejné rozložení jako má binární strom. Díky tomu se bude moci SKL se zadanými stromovými strukturami porovnávat.

Vyhledávání v SKL je obdobné jako v jednosměrně vázaném lineárním seznamu. Předpokládejme, že prvky v každé úrovni jsou seřazeny právě vzestupně. Při vyhledávání prvku platí, že se prvek začíná vyhledávat (horizontálně) v nejrychlejší lince v nejvyšší úrovni. Dokud další uzel v téže úrovni je větší jak hledaný klíč, hledáme v téže úrovni. Pokud je už následník menší jak hledaný uzel, sestoupíme na nižší úroveň (vertikálně). Tyto principy opakujeme, dokud nenalezneme hledaný prvek nebo nezjistíme, že ani na úrovni 1 se hledaný prvek nenachází. Pravděpodobnost, že při vyhledávání se budeme přesouvat horizontálně, je dána hodnotou $1 - p$, pravděpodobnost přesunu vertikálně je dána hodnotou p.

Pravděpodobnostní volba výšky

Nejdříve uvažme tuto situaci, která je zakreslena v 2.12. Nechť každý 2^i-tý prvek má ukazatel, který ukazuje o 2^i prvků vpřed, kde i je úroveň. Díky tomu je pak vyhledání konkrétního uzlu zredukováno na logaritmickou složitost. Čas vyhledání je nejvýše $\log_2 n$, kde n je počet prvků. Úroveň uzlu pak není třeba uchovávat, je to dáno počtem ukazatelů daného prvku.



Obrázek 2.12: Ideální přeskakující seznam – každý 2^i-tý prvek má ukazatel, který ukazuje o 2^i prvků dopředu

Tento princip má však jednu trhlinu. Pokud bychom měli vkládat či mazat prvek v této struktuře, museli bychom pro uchování tohoto uspořádání přerovnávat všechny úrovně následujících prvků za nově vloženým/smazaným prvkem, neboli provádět restrukturalizaci SKL. Proto se tato situace řeší pravděpodobnostním způsobem určování výšky jen u každého nově vkládaného prvku, kde úroveň prvku bude čistě stanovena dle hodnoty vygenerované rychlým a kvalitním pseudogenerátorem náhodných čísel o následujícím rozložení. Nechť jsou prvky do jednotlivých úrovní rozloženy s pravděpodobností $1/2^i$ (obecně p^i), kde i je úroveň. Pak opět bude při dostatečném počtu prvků dosaženo logaritmické složitosti vyhledávání, které je užíváno i ve vkládání a mazání. Jinými slovy požadujeme, aby prvek padl do první úrovně s pravděpodobností 50%, do druhé s 25%, do třetí s 12,5%, ... Vygenerovanou úroveň si prvek s sebou ponese po celou svou existenci. Takto navržená struktura je kopií běžných stromů s logaritmickou složitostí. Zavedením pravděpodobnosti jsou operace vkládání a ma-

zání (odstraňování) prvků v SKL ušetřeny od potřebné restrukturalizace, která je časově náročná. Ovšem tím, že každému prvku přiřadíme úroveň na základě pravděpodobnosti, je nezbytné předem znát počet úrovní, do kterých dané prvky můžeme distribuovat. Tento počet úrovní je neměnný po celou existenci SKL.

Vhodný výběr maximální úrovně (výšky) není jednoduchý a měl by být volen s rozvahou vzhledem k použití. Bude-li výška příliš vysoká, stráví se mnoho času při vyhledávání prvku, který pak je oproti současnému umístěn velice nízko. Tím by zisk struktury byl znehodnocen. Horní limit je i z tohoto důvodu nutný. Naneštěstí výška může být i příliš malá. K této situaci dojde při výrazném přeplnění struktury vysokým množstvím prvků. Došlo by tak k utvoření na každé úrovni moc dlouhého seznamu. Platí jednoduché pravidlo. SKL s maximálním počtem i úrovní by měl mít přibližně $(1/p)^i$ uzlů, neboli ideální stav volby úrovně je právě $log_{1/p} n$, kde n je očekávaný počet prvků v SKL (a p je zvolená pravděpodobnost jevu, viz dříve). V praxi lze pak považovat za rozumný počet úrovní v rozsahu $16-24$ pro většinu skutečných aplikací.

Protože práce v SKL spoléhá na pravděpodobnostní rozložení, struktura této vyhledávací metody není vhodná pro ukládání malého počtu prvků. Lze očekávat, že jednodušší a méně sofistikované algoritmy než je SKL budou dosahovat rychlejších výsledků, protože neprovádí tolik operací. Výhodou je, že v SKL neexistuje žádná vstupní sekvence vkládaných prvků generující nejhorší výkonnostní případ. Protože generované rozložení zajišťuje optimální rozložení přeskakujících ukazatelů a zcela nezávisí na pořadí žádaných prvků, můžeme ji řadit jako rovnocenného partnera ke stromovým strukturám. Pro mnoho aplikací může být SKL přirozenější strukturou než jsou stromy. Jednoduchost algoritmu umožňuje i jednoduchou implementaci. Za nedostatek lze však považovat, že SKL potřebuje předem znát přibližný počet dat vkládaných do této struktury, aby šlo stanovit počet úrovní, do kterých budou prvky rozmísťovány.

2.2 Implementace vybraných metod vyhledávání

Při výběru vhodného implementačního jazyka byl zvolen jazyk C, který je znám svým procedurálním přístupem (procedurální paradigma). I když by se dalo v implementaci uvažovat o použití jiného přístupu např. objektového, který je v současnosti čím dál tím více upřednostňován, byl použit programovací jazyk právě z procedurální skupiny jazyků, protože pohlíží na program jako na datové struktury a algoritmy. Ty jsou bližší k pojetí tématu práce nežli např. objekty a zprávy v objektově orientovaném přístupu. Volba právě na jazyk C padla z důvodu jeho širokého rozšíření a podrobného probírání již od začátku studia na této fakultě. Lze ho tak považovat za základní jazyk, který je každému dobře známý a srozumitelný, což nikdy neuškodí.

Ve všech probíraných stromových strukturách je každá uvažovaná operace (vkládání, hledání, mazání) v základu totožná s binárním vyhledávacím stromem. Až na vyvažování i z praktického hlediska jejich tvorby obtížnost zůstává zachována u všech operací jako v binárním vyhledávacím stromu. Tak jak je v binárním vyhledávacím stromu běžné, vkládání a hledání jsou jednoduché, obtížnější bývá odstraňování uzlu, nejedná-li se o listový uzel nebo o uzel, mající jen jediného přímého potomka (uzel mající nejvýše jeden podstrom). Ruší-li se uzel mající oba přímé potomky (oba podstromy), přepisuje se takový uzel kompletní kopií nejpravějším uzlem z levého podstromu nebo nejlevějším uzlem z pravého podstromu a odstraní se skutečně uzel ten, z kterého jsme kopírovali (zaručeně má nejvýše jeden podstrom). Cílem práce však není podrobně vysvětlovat základní operace nad binárním vyhledávacím

stromem, ale nad strukturami zadaných vyhledávacích metod. Proto se dále předpokládá, že binární vyhledávací strom je dostatečně znám a v popisu implementace se zaměříme na vyvažování nebo specifické úpravy těch částí stromů, které jsou odlišné od binárního vyhledávacího stromu. Totéž lze říci o jediné zadané nestromové struktuře přeskakujícího seznamu. Taktéž se očekává, že jednosměrně vázaný lineární seznam je znám na takové úrovni, že stačí se jen podrobně zaměřit na odlišnosti od základní struktury.

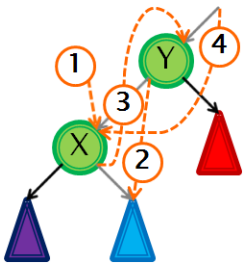
V této části budou uváděny různé významné a zajímavé části zdrojových kódů. Pokud se kdekoli v textu narazí při popisu stromu na ukazatel `T` nebo při popisu seznamu na ukazatel `L`, myslí se tím ukazatel na položku, do které jsou ukládány všechny důležité a společné informace pro celý strom či seznam. Příkladem může být ukazatel na absolutní kořen stromu nebo zcela první prvek seznamu tvořené struktury. Na existenci a očekávanou dostupnost těchto ukazatelů nebude dále upozorňováno.

2.2.1 Implementace AVL stromu (iterativním způsobem)

AVL strom se od binárního vyhledávacího stromu (dále BVS) nejvíce odlišuje svým vyvažováním, prováděné za pomoci různých rotací. Jak už bylo dříve uvedeno, každá rotace je jen cyklickou záměnou ukazatelů tak, aby se neporušila pravidla BVS a došlo k opětovnému vyvážení AVL stromu. Vzhledem k zrcadlové souměrnosti, budou z rotací popsány jen rotace LL a DLR.

U základních operací dochází k nutnosti provést kontrolu a vyvažování jen u vkládání a mazání, hledání je naprosto stejné jako v BVS. Proto se v dalším popisu dále zaměříme také na operace vkládání a mazání.

Rotace LL Popis procedury provádějící LL rotace (příslušné schéma v 2.13):



Obrázek 2.13: Záměna ukazatelů při LL rotaci

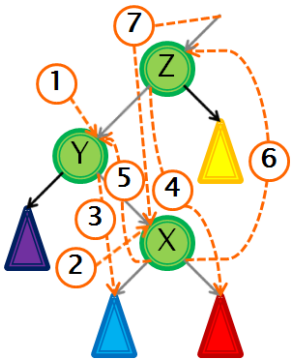
```
void rotateLL(tAVLNode*node, tAVLNode*nodeParent, tAVLTree*T)
{
  tAVLNode *nodeSuccLeft = node->left; // (1)
  node->left = nodeSuccLeft->right; // (2)
  nodeSuccLeft->right = node; // (3)
  if(nodeParent == NULL) // uzel Y je kořenem
    T->root = nodeSuccLeft; // (4)
  else{ // uzel Y má svého rodiče
    if(nodeParent->left == node) // Y je připojen zleva
      nodeParent->left = nodeSuccLeft; // (4)
    else // uzel Y je připojen ke svému rodiči zprava
      nodeParent->right = nodeSuccLeft; // (4)
  }
```

Před LL rotací bude mít kritický uzel vyvažovací faktor -2 a jeho levý potomek zaručeně -1 nebo 0 (jinak by to nebyla LL rotace). Provedeme tedy aktualizaci vyvažovacích faktorů (jejich hodnoty jsou odvozeny ve schématech v 2.3(a) na str. 9 a 2.4(a) na str. 10).

```
// levý potomek těžší vlevo
if(nodeSuccLeft->weight == -1){
  nodeSuccLeft->weight = 0; // uzel X
  node->weight = 0; // uzel Y
}
```

```
  // levý potomek výškově vyvážen
  else{
    nodeSuccLeft->weight = +1; // uzel X
    node->weight = -1; // uzel Y
  }
} // rotateLL()
```

Rotace DLR Popis procedury provádějící DLR rotace (příslušné schéma v 2.14):



Obrázek 2.14: Záměna ukazatelů při DLR rotaci

```
void rotateDLR(tAVLNode*node, tAVLNode*nodeParent, tAVLTree*T)
{
/* Vzhledem k tomu,
že záměna ukazatelů probíhá obdobným způsobem jako
v LL rotaci, věřím, že si každý domyslí, jak provést
úpravy propojení uzlů při této rotaci. Ukázka možného
postupu jejich úprav propojení je znázorněna v 2.14. */
```

Před DLR rotací bude mít kritický uzel vyvažovací faktor -2 a jeho levý potomek zaručeně $+1$ (jinak by to nebyla DLR rotace). Po úpravě ukazatelů je potřeba ještě aktualizovat faktory vyvážení příslušných uzlů (jejich hodnoty jsou odvozeny ve schématech v 2.3(b) na str. 9 a 2.4(b) na str. 10).

```
  if(nodeSuccLeftRight->weight == -1){ // těžší vlevo
    nodeSuccLeft->weight = 0; // uzel Y
    nodeSuccLeftRight->weight = 0; // uzel X
    node->weight = 1; // uzel Z
  }
  else if(nodeSuccLeftRight->weight == +1){ // těžší vpravo
    nodeSuccLeft->weight = -1; // uzel Y
    nodeSuccLeftRight->weight = 0; // uzel X
    node->weight = 0; // uzel Z
  }
  else{ // je výškově vyváženo
    nodeSuccLeft->weight = 0; // uzel Y
    nodeSuccLeftRight->weight = 0; // uzel X (toto lze vynechat)
    node->weight = 0; // uzel Z
  }
} // rotateDLR()
```

Vložení uzlu do AVL stromu

Operace vložení uzlu do stromu se dá rozdělit na 2 části. Jsou jimi vytvoření a sestavení uzlu, vyhledání jeho umístění a napojení do stromu, a pak část přepočtu vyvažovacích faktorů a provedení případné rotace.

První část je téměř stejná jako v binárním stromu. Jediný rozdíl je v tom, že v první části je navíc nastaven pomocný ukazatel na uzel (využit ve druhé části), od kterého se má provádět přepočet vyvažovacích faktorů. Je zřejmé, že to bude někde na cestě od kořene (včetně) k vloženému uzlu. Pokud se uzel do stromu vloží (klíč uzlu se ve stromu nenacházel),

je tento ukazatel nastaven na *nejvzdálenější* nakloněný uzel $(-/+1)$ od kořene stromu. Pokud v cestě ke vkládanému uzlu se žádný takový nakloněný uzel nenachází, je tento ukazatel nastaven na kořen. Význam však tohoto pomocného ukazatele je dvojí. Je to sice místo, od kterého se ve druhé části začínají směrem ke vloženému uzlu upravovat váhy uzlů, ale je to i *jediný potenciální kritický uzel*, nad kterým se pak bude volat rotace, pokud nedošlo k jeho úpravě na druhou stranu než byl nakloněn. Pokud z jakýchkoliv příčin žádný uzel nebyl do stromu vložen (např. uzel o vkládaném klíči se již ve stromu nacházel), do další části se nevstupuje a pomocný ukazatel je nenastavený. Toho může být využito např. pro rozhodnutí, zda do další části vstoupit.

V druhé části se tedy provádějí aktualizace vyvažovacích faktorů a volá se případná (jediná) rotace, pokud je potřeba. Popis této části je následující:

```
// oprava vyvážení AVL stromu po vložení nového uzlu

// nodeBalance je ukazatel na uzel, od něhož upravit vyvažovací faktor
node = nodeBalance; // pomocný ukazatel na uzel
while(node != NULL){ // úprava vyvažovacího faktoru
  // porovnání klíče vloženého uzlu s klíčem v aktuálním uzlu
  comparation = (*T->cmp)(Key, node->key);
  if(comparation < 0){
    node->weight -= 1; // uzel je vlevo o jeden přidaný uzel vyšší
    node = node->left; // posun vlevo
  }
  else if(comparation > 0){
    node->weight += 1; // uzel je vpravo o jeden přidaný uzel vyšší
    node = node->right; // posun vpravo
  }
  else{
    node->weight = 0; // jsme na vloženém uzlu
    node = NULL; // konec přepočítávání
  }
}
// kontrola zda třeba volat rotaci
if(nodeBalance->weight < -1){ // (stačilo by == -2)
  if((*T->cmp)(Key, nodeBalance->left->key) <= 0)
    // vložený uzel je vlevo od levého potomka kritického uzlu
    rotateLL(nodeBalance, nodeBalanceBack, T); // rotace zleva-zleva
  else // vložený uzel je vpravo od levého potomka kritického uzlu
    rotateDLR(nodeBalance, nodeBalanceBack, T); // rotace zleva-zprava
}
if(nodeBalance->weight > 1){ // (stačilo by == 2)
  if((*T->cmp)(Key, nodeBalance->right->key) >= 0)
    // vložený uzel je vpravo od pravého potomka kritického uzlu
    rotateRR(nodeBalance, nodeBalanceBack, T); // rotace zprava-zprava
  else // vložený uzel je vlevo od pravého potomka kritického uzlu
    rotateDRL(nodeBalance, nodeBalanceBack, T); // rotace zprava-zleva
}
```

Mazání uzlu z AVL stromu

Operace mazání (odstranění) uzlu z AVL stromu se dá také rozdělit na 2 části. První část je obdobná mazání uzlu z binárního vyhledávacího stromu, která už sama je oproti vkládání do binárního vyhledávacího stromu o něco implementačně složitější. Druhá část, která provádí vyvažování, však na rozdíl od vyvažování v operaci vkládání potřebuje znát úplnou zpětnou cestu od mazaného uzlu ke kořenu. To je dáno tím, že může být potřeba provést vyvažování nad každým uzlem v této cestě. Přístupů, jak potomek zjistí svého předka (rodiče), je několik – buď přidáním do uzlu ukazatele na svého rodiče, nebo s využitím datové struktury zvané zásobník. První zmíněný způsob však implementačně zesložiťuje např. prováděné rotace, a pokud není ve většině operací potřeba tento údaj znát, tento postup se nevyplatí. Proto se nejčastěji v implementaci AVL stromu užívá přístup přes zásobník. V rekurzivní implementaci se toto i přímo nabízí. V iterativní implementaci AVL stromu, kterou se tu zabýváme, je však nutno si tento zásobník sám explicitně vytvořit. Práce s tímto zásobníkem pak vypadá tak, že v první části do tohoto zásobníku si ukládáme ukazatele na projité uzly (cestu) od kořene k mazanému uzlu, v druhé části při vyvažování pak tyto ukazatele získáváme zpět v opačném pořadí, ve kterém jsme je do této struktury vložili (vlastnost LIFO), neboli známe vždy rodiče k aktuálnímu uzlu. Abychom se vyhnuli zbytečnému testování na zjištění, zda jsme právě v uzlu připojeném zleva či zprava k již známému rodiči, budeme si do tohoto zásobníku při vkládání ukazatelů (první část) ukládat i směr, kam jsme se z tohoto uzlu vydali k mazanému uzlu. Proces vyvažování po smazání uzlu je pak následující:

```
enum direction{NONE, LEFT, RIGHT}; // výčet (příznaky) směrů

// oprava vyvážení AVL stromu po smazání uzlu

// struktury obsahující ukazatel a směr (plněné ze zásobníku)
struct AVLNodeDirection NodeDirection, NodeDirectionParent;
// načtení rodiče skutečně odstraňovaného uzlu (potencionální kritický uzel)
// není-li rodič (zásobník je prázdný), očekává se, že se vrátí {NULL, NONE}
NodeDirection = topPopNodeDirection(Stack);
int end = 0; // příznak na ukončení cyklu vyvažování stromu

// cyklit, dokud není zásobník prázdný a nevyžádám si konec
while(NodeDirection.node != NULL && !end){
  // načtení prarodiče smazaného uzlu (rodiče možného kritického uzlu)
  NodeDirectionParent = topPopNodeDirection(Stack);
  if(NodeDirection.dir == LEFT){ // odebrali jsme uzel vlevo
    /* obdobné následujícímu */
  }
  else{ // odebrali jsme uzel vpravo
    NodeDirection.node->weight -= 1; // výška tohoto uzlu se zvýší vlevo
    switch(NodeDirection.node->weight){
      // uzel byl výškově vychýlen vpravo, odebráním uzlu vpravo
      // dojde k vyvážení tohoto uzlu; výška stromu s tímto uzlem se sníží,
      // není-li to kořen celé struktury, nutno přepočítat výšku o uzel výš
      case 0:
        if(NodeDirection.node == T->root){ // již skončit
          // (toto už mimochodem zajistí i prázdný zásobník,
          // zde přidáno jen pro přehlednost)
          end = 1;
```

```
        }
        break;
      // uzel byl výškově vyrovnán, odebráním uzlu vpravo
      // dojde k vychýlení tohoto uzlu vlevo; výška stromu se však zachová
      case -1:
        end = 1; // jsme s vyvažováním hotovi
        break;
      // uzel NodeDirection.node se stává kritickým uzlem
      case -2: // nutné provést rotaci
        if(NodeDirection.node->left->weight == +1){ // případ dvojité rotace
          // při dvojité rotaci se vždy sníží výška stromu s kořenem
          // NodeDirection.node (o -1 úroveň), nelze ukončit
          // přepočítávání výšky u svého rodiče
          rotateDLR(NodeDirection.node, NodeDirectionParent.node, T);
          // pokud rodič kritického uzlu neexistuje, předává se
          // ze zásobníku NULL, což je v souladu s touto rotací
        }
        else{ // případ jednoduché rotace
          if(NodeDirection.node->left->weight == -1){
            // případ jednoduché rotace s nevyváženým levým potomkem
            // kritického uzlu; vždy se sníží výška stromu
            // s kořenem NodeDirection.node, nelze ukončit vyvažování
            rotateLL(NodeDirection.node, NodeDirectionParent.node, T);
            // pokud rodič kritického uzlu neexistuje, předává se
            // ze zásobníku NULL, což je v souladu s touto rotací
          }
          else{ // případ jednoduché rotace s vyváženým levým potomkem
            // kritického uzlu; nedojde již ke snížení stromu
            // s kořenem NodeDirection.node, ukončit vyvažování
            end = 1; // již ukončit cyklus vyvažování
            rotateLL(NodeDirection.node, NodeDirectionParent.node, T);
            // pokud rodič kritického uzlu neexistuje, předává se
            // ze zásobníku NULL, což je v souladu s touto rotací
          }
        }
        break;
    } // switch()
  } // else
  // posun o úroveň blíže ke kořenu celé AVL struktury
  NodeDirection = NodeDirectionParent;
} // while()
```

2.2.2 Implementace červeno-černého stromu

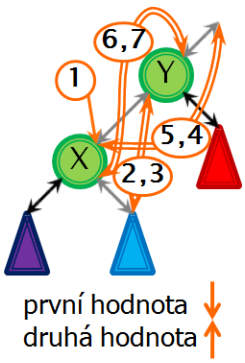
V algoritmu červeno-černého stromu (dále RBT) při vyvažování stromu je základním požadavkem, aby uzel znal svého rodiče. To, jak bylo popsáno v implementaci AVL stromu, lze provést buď za pomoci zásobníku, nebo přidáním ukazatele do struktury uzlu. Protože však při vyvažování RBT více uzlů neležících ve stejné cestě zároveň potřebuje znát svého

rodiče, použití zásobníku je prakticky nemožné a bylo by málo efektivní. Je proto potřeba, aby struktura uzlu v sobě obsahovala ukazatel na svého rodiče. To sice implementačně znepříjemní provádění všech rotací, ale výrazně ulehčí pak vyvažování RBT stromu.

Z dalších odlišností RBT je to, že každý uzel má svůj barevný atribut. Ten se ukládá do uzlu, protože je to jeho charakteristická vlastnost tak jako vyvažovací faktor v AVL uzlu. Důležité je, že za list stromu v RBT je považován až *NULL* uzel, který je černý. Aby se vyhnulo potřebě zbytečného testování a rozlišování, zda daný uzel je či není *NULL* pro zjištění, zda lze v odkazované hodnotě hledat např. barevný atribut či ne, byl utvořen speciální uzel `myNULL`, který tento barevný atribut v sobě obsahuje. U procesu vyvažování po smazání uzlu se zašlo ještě dále. Očekává se, že tento listový uzel má i ukazatel na svého rodiče (viz teoretická část). Proto pro jednoduchost práce s tímto uzlem jako se skutečným bylo nakonec zvoleno, že obsahuje všechny položky jako pravý uzel. Samozřejmě že barva je mu nastavena na černou. Ostatní ukazatele jsou sice nastaveny na *NULL*, ale nemá to žádný teoretický význam (jen praktický pro ladění chybného programu, protože program při náhodné dereferenci takovéhoto ukazatele okamžitě spadne). Aby nedocházelo ke značnému ztrátovému prostoru, `myNULL` byl vytvořen právě jeden, na který je vždy odkazováno, pokud daná větev stromu končí. To si lze dovolit díky tomu, že `myNULL` uzel si musí naštěstí pamatovat nejvýše jen jednoho konkrétního rodiče, který je mu vždy před použitím nastaven (případ smazání vnitřního listu).

V ostatních oblastech tento strom již není nijak zvláštně specifický. U RBT dochází k potřebě provádět vyvažování jen při vkládání a mazání, hledání je naprosto stejné jako např. v AVL stromu nebo binárním vyhledávacím stromu (dále BVS). Proto se v dalším popisu zaměřme na vyvažování těchto operací, před tím velice krátce např. na tvorbu rotace doprava z důvodu řečeného ztížení implementace přidáním ukazatele na rodič uzlu.

Rotace doprava Popis procedury provádějící rotaci doprava (příslušné schéma v 2.15, první hodnota (bráno zleva) popisuje propojení rodiče na potomka, druhá z potomka na rodiče):



Obrázek 2.15: Záměna ukazatelů při rotaci doprava

```
void rotateRight(tRedblackTree *T, tRBTNode *node)
{
  tRBTNode *nodeSuccLeft = node->left; // (1)
  node->left = nodeSuccLeft->right; // (2)
  // pokud pravý podstrom uzlu X není list
  if(nodeSuccLeft->right != myNULL)
    nodeSuccLeft->right->parent = node; // (3)
  nodeSuccLeft->parent = node->parent; // (4)
  if(node->parent == myNULL) // uzel Y je kořenem
    T->root = nodeSuccLeft; // (5)
  else{
    // Y je připojen zprava
    if(node == node->parent->right)
      node->parent->right = nodeSuccLeft; // (5)
    else // jinak uzel Y je připojen ke svému rodiči zleva
      node->parent->left = nodeSuccLeft; // (5)
  }
  nodeSuccLeft->right = node; // (6)
  node->parent = nodeSuccLeft; // (7)
```

```
} // rotateRight()
```

Jak je vidět, musí se dávat jen pozor na všechny ukazatele, na žádný nezapomenout. U přiřazování ukazatele okrajovému uzlu (ať již směrem k listu nebo kořenu stromu) se nesmí zapomenout nejdříve ověřit, zda tento uzel není koncový. Na rozdíl u rotace nastíněné u AVL stromu, která zároveň aktualizovala vyvažovací faktory, u RBT se jedná pouze o holou rotaci, správa korektního nastavení barvy uzlu je přenechána proceduře, která tuto rotaci volala.

Vložení uzlu do RBT

Operace vložení uzlu do RBT se dá rozdělit na 2 části – vložení nového uzlu do RBT, které probíhá zcela stejně jako do BVS, a vyvažování RBT. Jediné, co je potřeba pro vyvážení stromu znát, je platný ukazatel na nově vložený uzel, který má vždy červenou barvu. Od něj se provádí vyvažování. Vzhledem k tomu, že jednotlivé případy byly už podrobně popsány v teoretické části, následuje komentovaná stěžejní část operace vyvažování při vložení nového uzlu.

```
// oprava vyvážení RBT po vložení nového uzlu

// node je ukazatel nastavený na nově vložený uzel, od něhož může být
// porušen RBT strom, tento ukazatel postupně posouváme ke kořenu stromu
tRBTNode *node = ···;

tRBTNode *nodeParentOtherSide; // pomocný ukazatel na strýce uzlu node
node->colour = RED; // vložený uzel je vždy červený

// vyrovnávat, dokud ukazatel node neukazuje na kořen RBT stromu
// a rodič uzlu, na který se tímto ukazatelem ukazuje, je také červený;
// v každé iteraci ukazatel node lze interpretovat jako ukazatel,
// od něhož dolů k listu je vše vyrovnáno dle pravidel RBT
while(node != T->root && node->parent->colour == RED){
  // rodič je levý potomek prarodiče (existenci prarodiče zajišťuje
  // vždy nastavení kořene struktury na černý po ukončení cyklu)
  if(node->parent == node->parent->parent->left){
    // nastavíme si pomocného ukazatele na strýce uzlu node
    nodeParentOtherSide = node->parent->parent->right;
    // pokud existuje teta/strýc a má barvu červenou
    // (neexistuje-li, je myNULL, který se považuje za černý)
    if(nodeParentOtherSide != myNULL && nodeParentOtherSide->colour == RED){
      node->parent->colour = BLACK; // případ 1
      nodeParentOtherSide->colour = BLACK; // případ 1
      node->parent->parent->colour = RED; // případ 1
      node = node->parent->parent; // případ 1
    }
    else{ // teta/strýc uzlu node je černý
      // je-li vložený uzel (v této levé verzi) vložen vpravo
      if(node == node->parent->right){
        node = node->parent; // případ 2
        rotateLeft(T, node); // případ 2
      }
```

```
      // node je od rodiče vlevo (jeho teta/strýc je černý)
      node->parent->colour = BLACK; // případ 3
      node->parent->parent->colour = RED; // případ 3
      rotateRight(T, node->parent->parent); // případ 3
    } // else
  } // if()
  // rodič je pravý potomek prarodiče (existenci prarodiče zajišťuje
  // vždy nastavení kořene struktury na černý po ukončení cyklu)
  else{
    /* pravá verze, symetrická k větvi if, všude prohodit left a right */
  }
} // while()
T->root->colour = BLACK; // kořen je vždy černý
```

Mazání uzlu z RBT

Operace mazání (odstranění) uzlu z RBT se dá také rozdělit na 2 části – odstranění jistého uzlu (bez ohledu na pravidla RBT) a následně zavolání procedury na vyvážení RBT. První část je téměř stejná jako odstranění uzlu v BVS. Odlišnost tkví jen v tom, že při odstranění uzlu z RBT se přiřazuje jeho (jedinému) potomkovi ukazatel na rodiče odstraňovaného uzlu bez ohledu na to, že tento potomek může být `myNULL`. Vyvažování stromu se volá jen tehdy, byl-li odstraněný uzel černé barvy, protože jinak se žádné pravidlo neporuší. Proceduře vyvažování se předává ukazatel na potomka odstraněného uzlu. Potomek může být jak vnitřní uzel RBT, tak i listový uzel. Důležité však je, že má správně nastavený ukazatel na svého nového rodiče. Protože jednotlivé případy, které je třeba řešit při odstraňování uzlu, byly už podrobně probrány v teoretické části, dále následuje jen komentovaná stěžejní část operace procesu vyvažování při vymazání/odstranění uzlu.

```
// oprava vyvážení RBT po smazání uzlu

// lackBlackNode je ukazatel nastavený na uzel, který má 2 černé barvy,
// neboli je nastavený na uzel, v jehož cestě chybí černý uzel
tRBTNode *lackBlackNode = ···;

// ukazuje-li lackBlackNode na červený uzel, přebarvíme ho za cyklem
// na černo; ukazuje-li lackBlackNode na uzel kořene celé struktury,
// lze na nedostatek černých uzlů zapomenout; jinak potřeba provést nějakou
// rotaci a přebarvení uzlů (vstup do cyklu)
while(lackBlackNode != T->root && lackBlackNode->colour == BLACK){
  tRBTNode *sibling; // pomocný ukazatel na sourozence
  // je-li uzel s nedostatečným počtem černé vlevo od svého rodiče
  if(lackBlackNode == lackBlackNode->parent->left){
    // nastavení ukazatele na sourozence - určitě není myNULL,
    // protože lackBlackNode chybí černý uzel
    sibling = lackBlackNode->parent->right;
    if(sibling->colour == RED){ // sourozenec je červený
      sibling->colour = BLACK; // případ 1
      lackBlackNode->parent->colour = RED; // případ 1
      rotateLeft(T, lackBlackNode->parent); // případ 1
      sibling = lackBlackNode->parent->right; // případ 1
```

```
    }
    // sourozenec je černý, má právě oba černé potomky (můžou být myNULL)
    if(sibling->left->colour == BLACK && sibling->right->colour == BLACK){
      sibling->colour = RED; // případ 2
      lackBlackNode = lackBlackNode->parent; // případ 2
    }
    else{ // sourozenec je černý, jeho potomci nejsou oba černí
      // sourozenec je černý, jeho pravý potomek je černý (lze být myNULL)
      // a jeho levý potomek je červený (levý zaručeně není myNULL)
      if(sibling->right->colour == BLACK){
        sibling->left->colour = BLACK; // případ 3
        sibling->colour = RED; // případ 3
        rotateRight(T, sibling); // případ 3
        sibling = lackBlackNode->parent->right; případ 3
      }
      // sourozenec je černý, jeho pravý potomek je červený
      // a jeho levý potomek je libovolný
      sibling->colour = lackBlackNode->parent->colour; // případ 4
      lackBlackNode->parent->colour = BLACK; // případ 4
      sibling->right->colour = BLACK; // případ 4
      rotateLeft(T, sibling->parent); // případ 4
      lackBlackNode = T->root; // nepodmíněné ukončení cyklu
    } // else
  } // if()
  else{ // je-li uzel s nedostatečným počtem černé vpravo od svého rodiče
    /* pravá verze, symetrická k větvi if, všude prohodit left a right */
  }
} // while()
// je-li uzel červený, chybí v jeho podstromu černý uzel, obarvíme ho černě
// (byl-li to kořen, je vždy černý a případné přebarvení na totéž nepoškodí)
lackBlackNode->colour = BLACK;
```

2.2.3 Implementace rozvinutého stromu

Vzhledem k tomu, že práce předpokládá znalost binárního vyhledávacího stromu, který je základem rozvinutého stromu (dále SPT), je z implementačního hlediska zajímavá pouze splay operace. Jak jsme se dozvěděli z teoretické části, existují dvě její modifikace – bottom-up splay a top-down splay.

Bottom-up splay

Bottom-up splay užívá v podstatě dvojí průchod stromem tak, že nejdříve nalezne uzel, který umístit do kořene, a pak jej přesune do kořene stromu s využitím dříve znázorněných rotací (viz 2.1, str. 5). Implementace rotací je podobná nebo v některých případech i stejná jako v předchozích metodách, a proto je zbytečné si je tu blíže uvádět. Důležité ale je, že jednotlivé rotace se aplikují na uzly v opačném pořadí jak byly projity. Je tedy nutné nějakým způsobem znát zpětnou cestu, což nebývá samozřejmé. I když lze zvolit více způsobů pro jejich znalost, následující ukázka předpokládá explicitně vytvořený a již naplněný zásobník,

protože v implementaci všech předchozích metod byly upřednostněny jejich iterativní formy realizace.

```
// ukázka bottom-up splay

··· // vytvoření a naplnění zásobníku cestou z kořene do přesouvaného uzlu
int comp1; // příznak porovnání (dle klíčů): přesouvaný uzel-rodič
int comp2; // příznak porovnání (dle klíčů): přesouvaný uzel-prarodič
tSPTNode *grandparent, *parent; // ukazatelé na uzel prarodiče a rodiče
T->root = node; // nastavení ukazatele na uzel, který se přesouvá do kořene

// dokud v zásobníku máme 2 a více ukazatelů na rodiče a prarodiče
while(stack.size > 1){ // dvojitá rotace
  parent = topPopNodePtr(&stack); // poznačení si ukazatele na rodiče
  grandparent = topPopNodePtr(&stack); // a ukazatele na prarodiče
  comp1 = (*T->cmp)(Key, parent->key); // porovnání klíče s klíčem rodiče
  comp2 = (*T->cmp)(Key, grandparent->key); // a klíče s klíčem prarodiče
  if(comp1<0 && comp2<0){ ··· /* zig-zig rotace */ }
  else if(comp1>0 && comp2>0){ ··· /* zag-zag rotace */ }
  else if(comp1>0 && comp2<0){ ··· /* zig-zag rotace */ }
  else if(comp1<0 && comp2>0){ ··· /* zag-zig rotace */ }
}
if(stack.size == 1){ // existuje jen rodič -> jednoduchá rotace
  parent = topPopNodePtr(&stack); // poznačení si ukazatele na rodiče
  comp1 = (*T->cmp)(Key, parent->key); // porovnání klíče s klíčem rodiče
  if(comp1 < 0){ ··· /* zig rotace */ }
  else{ ··· /* zag rotace */ }
}
```

Top-down splay

Top-down splay prochází strom jen jednou s použitím 3 stromů. Prostřednímu stromu, který na počátku obsahuje celý SPT, jsou neustále odřezávány jednotlivé větve a přidávány do levého nebo pravého stromu, dokud se v kořenu prostředního stromu nenajde hledaný uzel (nebo kořen prostředního stromu nemá už očekávaným směrem žádného potomka). Poté dojde k sestavení všech stromů do jediného tak, že levý a pravý strom se připojí (každý jako jedna větev) k prostřednímu stromu. Protože principy jednotlivých úprav byly už názorně zakresleny v 2.9 (str. 20), nepovažuji za významné tu rozepisovat tyto elementární úpravy. Místo toho se podívejme, jak top-down splay operace tyto úpravy využívá.

```
// ukázka top-down splay

if(T->root == NULL) // je-li struktura prázdná, není co přesouvat do kořene
  return;

// kořeny levého, pravého a prostředního stromu
tSPTNode leftSubTree, rightSubTree, *middleSubTree;
// inicializace hlavičky levého kořene
leftSubTree.left = &leftSubTree;
leftSubTree.right = NULL;
// inicializace hlavičky pravého kořene
```

```
rightSubTree.left = NULL;
rightSubTree.right = &rightSubTree;
// inicializace prostředního stromu - na začátku obsahuje celý SPT strom
middleSubTree = T->root;

int comp; // pomocná proměnná s výsledkem porovnání
while(1){ // hledání žádaného uzlu (v prostředním stromu) + jeho rozřezávání
  // porovnáme hledaný klíč s klíčem kořene prostředního stromu
  comp = (*T->cmp)(Key, middleSubTree->key);
  if(comp < 0){ // je-li hledaný klíč menší jak klíč kořene
    if(middleSubTree->left){ // pokud má kořen levého potomka
      // porovnáme hledaný klíč s klíčem levého potomka kořene
      comp = (*T->cmp)(Key, middleSubTree->left->key);
      // hledaný klíč je menší jak klíč levého potomka kořene
      // -> rotace vpravo a připojení vpravo (jako zig-zig operace)
      if(comp < 0){
        middleSubTree = rotateRight(middleSubTree); // rotace vpravo
        // pokud uzel má levého následníka
        // (lze v zig-zig operaci pokračovat)
        if(middleSubTree->left){
          // připojení vpravo
          middleSubTree = linkRight(middleSubTree, &rightSubTree);
        }
        // uzel nemá levého následníka
        // (nelze v (zig-)zig operaci pokračovat)
        else{
          break; // požadovaný uzel se ve stromu nenachází
        }
      }
      // hledaný klíč je větší jak klíč levého potomka kořene
      // -> připojení vpravo a připojení vlevo (jako zig-zag operace)
      else if(comp > 0){
        // připojení vpravo
        middleSubTree = linkRight(middleSubTree, &rightSubTree);
        // pokud uzel má pravého následníka
        // (lze v zig-zag operaci pokračovat)
        if(middleSubTree->right){
          // připojení vlevo
          middleSubTree = linkLeft(middleSubTree, &leftSubTree);
        }
        // uzel nemá pravého následníka
        // (nelze v (zig-)zag operaci pokračovat)
        else{
          break; // požadovaný uzel se ve stromu nenachází
        }
      }
      // hledaný klíč roven klíči levého potomka kořene
      // -> připojení vpravo (jako zig operace)
      else{
```

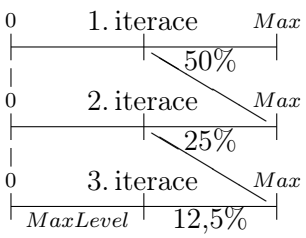
```
        // připojení vpravo
        middleSubTree = linkRight(middleSubTree, &rightSubTree);
      }
    }
    else
      break; // požadovaný uzel se ve stromu nenachází nebo byl již nalezen
  }
  else if(comp > 0){ // je-li hledaný klíč větší jak klíč kořene
    ··· // symetricky pravá verze
  }
  else // hledaný klíč je roven klíči kořene, není třeba hledat
    break;
}

// sestavení jednotlivých stromů
assemble(middleSubTree, &leftSubTree, &rightSubTree);
T->root = middleSubTree; // kořen sestaveného stromu se stává kořenem i SPT
```

2.2.4 Implementace přeskakujícího seznamu

Přeskakující seznam (dále SKL) se od jednosměrně vázaného lineárního seznamu odlišuje tím, že vytváří paralelně uspořádané jednosměrně vázané lineární seznamy (dále LL). Tyto paralelně uspořádané LL však ve skutečnosti neexistují, vyskytuje se jen jediný LL, nad kterým je tvořena úrovňová hierarchie propojení různých uzlů (prvků seznamu). Říkáme, že uzel je výšky i, pokud se již nenachází v úrovni $i + 1$. Nejnižší výška uzlu je 1 (jedná se o prostý prvek LL). Naopak nejvyšší povolenou hodnotou je hodnota, která je získána při inicializaci SKL. Označme ji dále jako $MaxLevel$. Tato hodnota nesmí být nikdy překročena. Za celkovou úroveň SKL je považována nejvyšší úroveň v aktuálním čase (pokud je seznam prázdný, lze ji považovat za nulovou). SKL užívá jako první položku seznamu hlavičku. Ta zná ukazatele na všechny povolené úrovně (od 1 do $MaxLevel$), které jsou na začátku inicializované do výchozí hodnoty ($NULL$).

Pseudogenerátor náhodných čísel



Obrázek 2.16: Znázornění průběhu tvoření žádaného rozložení pro 3 iterace (MaxLevel je 4)

Jak už bylo dříve řečeno, daný uzel má svoji úroveň (výšku) získanou z pseudogenerátoru náhodných čísel. Ten musí předávat hodnoty tak, aby každá vyšší hodnota byla vždy půlkrát pravděpodobnější (dvakrát méně pravděpodobná) jak následující hodnota[13]. Takové rozložení pravděpodobnosti však není příliš rozšířené. Proto se vytváří za pomoci běžně dostupného rovnoměrného pseudogenerátoru náhodných čísel, kde výstup jeho hodnot ztransformujeme do požadovaného rozložení a rozsahu $\langle 1, MaxLevel \rangle$ následovně:

```
// ukázka transformace do žádaného rozložení

int randHeight(int max)

{
```

[13]Obecně se požaduje, aby každá vyšší hodnota byla vždy p-krát pravděpodobnější, kde p je zvolená pravděpodobnost.

```
  int height = 1; // nejnižší úroveň/výška je vždy 1
  // polovina z rozsahu pseudogenerátoru
  int max_2 = RAND_MAX/2;
  while(height<max && rand()<max_2)
    ++height; // zvyš úroveň
  return height;
} // randHeight()
```

Jak ukazuje schéma 2.16 (str. 37) znázorněné pro 3 iterace, vrácená hodnota z této transformace je hledaná výška uzlu. Na pseudogenerátor je především kladen důraz na rychlost a rovnoměrnost rozložení velkých a malých hodnot.

Implementace operací pro vložení a smazání prvku

Díky tomu, že SKL užívá přeskakující ukazatele, realizace vložení a smazání prvku je o něco ztížena. U obou operací se principiálně provádí totéž co v LL s tím, že je nezbytné zároveň si značit všechny ukazatele ze všech úrovní, aby šlo vkládaný prvek do struktury napojit a mazaný prvek před odstraněním ze struktury odpojit. Protože se „pracuje“ jen s jednosměrně vázanými LL (víme, že jde jen o abstrakci vytvářenou ukazateli), které nemají možnost jednoduše si zjistit svého přímého předchůdce, je potřebné si v každé úrovni předem aktivně značit potenciální předchozí prvek od aktuálního prvku. Po nalezení vhodného místa pro vkládaný prvek nebo nalezení prvku ke smazání již tak budeme znát jejich aktuální předchůdce (vyhýbá se tak opětovnému průchodu v tomtéž seznamu). I když způsobů jak provést si toto značení je několik, pravděpodobně nejjednodušší způsob je vytvořit si pole ukazatelů na předchozí prvky, kde index pole nám bude reprezentovat úroveň[14]. Procházení však každé úrovně samostatně pro nalezení svého předchůdce by bylo neefektivní. Proto se užívá typického (dvojrozměrného) způsobu procházení hierarchií SKL dopředných ukazatelů. Popsaný způsob procházení je uveden v popisu vyhledávání v teoretické části SKL.

Aby nebylo potřeba ošetřovat často nadbytečnými testy problematiku neexistence žádného prvku v SKL, implementace užívá principu hlavičky (přímo se to i nabízí), která je vytvořena při inicializaci struktury. Hlavička SKL je slepý prvek o jeho maximální výšce, protože je potřeba si v ní uchovávat ukazatele na všechny úrovně SKL.

```
// ukázka uchování si ukazatelů na všechny předchozí prvky
// od vhodné pozice prvku (vložení) nebo od mazaného prvku (smazání)

tSKLNode *fix[L->max]; // pomocné pole úrovní (o rozměru max. výšky SKL)
// přes všechny úrovně aktuální výšky SKL (height) do úrovně 1
// (index snížen o 1 z důvodu indexování polí od 0)
for(int i=L->height-1; i>=0; i--){
  // nejsme-li na konci a nepřešli-li jsme (hledaný klíč > klíč uzlu)
  // (užito tzv. zkratové vyhodnocení)
  while(node->next[i]!=NULL && (*L->cmp)(Key, node->next[i]->key)>0)
    node = node->next[i]; // posun v úrovni i na další prvek
  fix[i] = node; // poznačení si předchozího prvku pro danou úroveň
}

// ukázka vložení (napojení) prvku do SKL

// tSKLNode *item je nově vkládaný prvek
```

[14] V jazyce C nechť index 0 znamená úroveň 1 atd.

```
// byla-li pro vkládaný prvek vygenerována úroveň vyšší
// než je nyní aktuální (nikoli maximální) úroveň SKL
if(item->actHeight > L->height){
  // zvýšit úroveň celé struktury, tuto hodnotu přiřadit vkládanému prvku
  item->actHeight = ++(L->height);
  // přidali jsme úroveň, poznačení, že hlavička se také určitě stává
  // předchozím prvkem v nejvyšší úrovni vkládaného prvku
  fix[item->actHeight-1] = L->head;
}
int h = item->actHeight;
// přes všechny úrovně aktuální výšky SKL (actHeight) do úrovně 1
// (index snížen o 1 z důvodu indexování polí od 0)
while(--h >= 0){
  item->next[h] = fix[h]->next[h]; // propojení vkládaného prvku s následným
  fix[h]->next[h] = item; // propojení předchozího prvku s vkládaným
}

// ukázka smazání (odstranění) prvku v SKL

// tSKLNode *item je mazaný prvek
// pro všechny ukazatele ukazující právě na mazaný prvek
for(int i=0; i<item->actHeight; i++){
  // přemostíme odstraňovaný prvek
  fix[i]->next[i] = fix[i]->next[i]->next[i];
}
while(L->height > 0){ // není-li SKL prázdný
  // SKL má ještě alespoň jeden prvek v nejvyšší úrovni
  if(L->head->next[L->height-1] != NULL)
    break;
  // na nejvyšší úrovni SKL už nic není,
  // úroveň v hlavičce zakončíme a snížíme
  L->head->next[--(L->height)] = NULL;
}
```

2.3 Knihovna vybraných metod vyhledávání

Ačkoli zadání práce vyžaduje pouze jednoduchou implementaci vybraných metod vyhledávání, považuji za nedostatečné, omezit se při tvorbě vyhledávacích metod jen na jejich implementaci pro jediné použití. Proto jsem se rozhodl v této oblasti dosáhnout vyššího cíle – vytvořit malou knihovnu zadaných metod vyhledávání tak, aby byla připravena pro užití i v dalších aplikacích. Proto je třeba si uvědomit, že v praxi budou v uzlech[15] uloženy aplikací vyžadovaná data. Navíc není předem známo, že klíč uzlů je např. jednoduchého typu `int`. Kdybychom se pohybovali na úrovni jazyka C++, právě zde by byl vhodný případ pro použití šablon. Jak už ale bylo dříve řečeno, pro implementaci byl použit jen základní jazyk C, který možnosti šablon nemá. Jak tedy se přiblížit k síle šablon, které má C++?

Uvědomme si, že všechny tvořené algoritmy vyžadují jen porovnávací funkci nad určitým typem klíče. My však tento typ předem neznáme, a proto neznáme ani porovnávací funkci

[15] Následující text striktně nerozlišuje mezi pojmem uzel a prvek, při jejich užití se myslí obojí.

(relaci porovnání nad zadaným typem klíče). Nezbývá tedy nic jiného, než použít ukazatel nespecifikovaného typu na předem neznámý typ (`void *`) a ukazatel na předem neznámou implementaci porovnávací funkce s deklarací `int comp(const void *, const void *);`, což je naprosto standardní deklarace porovnávací funkce, která se vyskytuje i v knihovních funkcích jazyka C. Tak jak je běžné, tato uživatelem knihovny tvořená porovnávací funkce by měla na základě porovnání klíčů obdržených přes ukazatele vracet:

- zápornou hodnotu, je-li první argument menší jak druhý
- 0 (slovy nula), je-li první argument roven druhému argumentu
- kladnou hodnotu, je-li první argument větší jak druhý

S datovými položkami, které budeme požadovat uložit do uzlu, použijeme obdobnou věc jako pro klíč – použijeme ukazatel na datovou strukturu, která předem nebyla plně deklarována[16], pouze pojmenována (určeno jméno typu). Pro jednotlivé metody (AVL tree, red-black tree, splay tree a skip list) jsou požadována následné pojmenování struktur (v uvedeném pořadí): `struct AVLDataItem`, `struct RBTDataItem`, `struct SPTDataItem` a `struct SKLDataItem`. Díky ukazatelům je nám tedy umožněno vytvořit a zkompilovat knihovnu bez toho, aniž bychom předem znali program, ve kterém tuto knihovnu budeme chtít užít. Jinými slovy, za pomoci ukazatelů jsme posunuli požadavek znalosti konkrétního typu klíče, datových položek struktury dat ukládaných do uzlu a porovnávací funkci nad zadaným typem klíče až do tvorby programu, který bude chtít tuto knihovnu použít. Proto je pro užití této knihovny v tvořené aplikaci nezbytně nutné:

- definovat strukturu dat ukládaných do každého uzlu
- vytvořit porovnávací funkci nad požadovaným typem klíče
- předat správně v inicializaci datové struktury[17]:
  - velikost datového typu klíče (počet bajtů)
  - velikost struktury dat (počet bajtů)
  - a ukazatel na porovnávací funkci

Obraz každé vyhledávací metody je uložen jistým způsobem do struktury (sítě) různě provázaných uzlů. Protože může být požadavek na zobrazení tohoto provázání, knihovna umožňuje tuto strukturu vytisknout (zobrazit, vypsat) do zadaného datového streamu (proudu) předaného funkci tisku celé struktury. Protože však z pohledu knihovny není známo, co za případné datové položky se v uzlu nacházejí, je při volání funkce tisku (celé) struktury potřeba předat v jejím parametru ukazatel na uživatelem vytvořenou funkci tisku jediného uzlu. V parametrech této funkce se (pro čtení) předává datový proud, do kterého má tato funkce žádané položky vytisknout, a ukazatel na uzel, z kterého si bude uživatel knihovny přát žádané položky zobrazit. Tato funkce je volána pro každý výtisk uzlu celé struktury. Její deklarace je obdobná pro všechny vyhledávací metody, liší se jen typem uzlu dané metody. Např. pro AVL tree je její deklarace `void AVLPrintNode(FILE *, const tAVLNode *);`. Přehled operací, které vytvořená knihovna nabízí, nalezneme v tabulce 2.1 (str. 42). Z důvodu úspory místa neuvádím jejich parametry a návratové hodnoty. Jedná

[16] Jedná se o tzv. neúplnou deklaraci struktury, která lze použít pro definici ukazatelů, nikoli však pro definici proměnné.

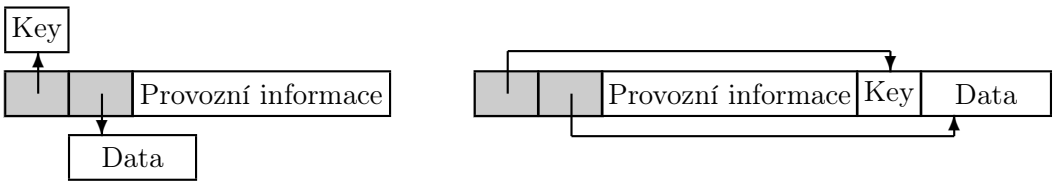
[17] Poznamenejme, že inicializace je zásadně první operací, která se nad danou operací vykonává.

se pouze o jejich názvy. Přesné deklarace a podrobný popis chování nalezneme v hlavičkovém souboru příslušné metody. Před užitím knihovny doporučuji přečíst si tyto podrobné komentáře. Jednoduchý příklad použití této knihovny uvádím v příloze.

I když se toto řešení může zdát složité, myslím si, že je to možná i jediné řešení, jak se v jazyce C vyhnout potřebě použití šablon, které v něm neexistují. Výsledkem tohoto úsilí, a to bych zdůraznil, je *zcela nezávislá knihovna na datech, která je schopná pracovat nad zcela libovolným typem klíče se zcela libovolnými datovými položkami ukládanými do uzlu.*

Vraťme se však ještě k inicializaci datové struktury a následně k vlastnímu paměťovému uspořádání uzlu. Řekli jsme, že klíč uzlu bude typově nespecifikovaný ukazatel, taktéž datová struktura uzlu nebude předem plně deklarována. Při inicializaci však uživatel této knihovny sám (správně) předá velikost typu klíče, velikost datové struktury (klíč a datová struktura se ukládají do každého uzlu) a ukazatel na porovnávací funkci (poznamenáno jen jednou do struktury stromu). Proč tyto požadavky? Protože díky znalosti velikosti jednotlivých položek a porovnávací funkce dosáhneme toho, že (z pohledu knihovny) nepotřebujeme znát žádnou interpretaci předaného klíče ani obdržených dat, i když obojí potřebujeme uložit do uzlu. Interpretace klíče uzlu, dle kterého se do struktury ukládá, je pevně řízena dle návratové hodnoty porovnávací funkce. Teprve tato funkce zná interpretaci klíče uzlu, protože ji vytváří až uživatel knihovny. U obdržených dat je to ještě jednodušší, protože i z pohledu algoritmů je nám už naprosto jedno, co za typ a počet dat uživatel z uzlu čte nebo ukládá. Známe totiž od inicializace, jak budou data bajtově veliká. Při jejich ukládání do uzlu je přebereme a bez jakékoli interpretace uložíme někam do paměti uzlu, naopak při žádosti o ně, vezmeme opět tutéž velikost a předáme je uživateli knihovny zcela tak, jak byly obdrženy.

Bylo řečeno, že klíč i data ukládáme do uzlu. Nabízí se dvě varianty řešení, jakým způsobem alokovat paměť pro uzel (viz 2.17(a) a 2.17(b), str. 41). Buď alokovat vždy paměť pro klíč, data a provozní informace uzlu zvlášť, nebo si spočítat celkovou spotřebu paměti pro jeden uzel a alokovat paměť jako jeden celek. První řešení je programátorsky triviální, jedná se jen o alokaci 3 úseků paměti a jejich vzájemné propojení. Druhé řešení je náročnější v tom, že se musí (velmi) přesně spočítat pozice v jediném alokovaném prostoru, kam uložit klíč a data, na které se následně nastaví příslušné ukazatele. Jak názorně ukazuje tabulka těchto dvou návrhů 2.2 (str. 42) vytvořená na základě vlastního testu, druhý způsob je rychlejší. Proto každý uzel struktury v implementované knihovně je utvořen podle tohoto druhého způsobu.



(a) Návrh I (b) Návrh II

Obrázek 2.17: Způsoby alokace paměti pro uzel

	AVL tree	Red-black tree	Splay tree[18]	Skip list
Inicializace struktury	AVLInit()	RBTInit()	SPTInit()	SKLInit()
Uvolnění struktury	AVLDispose()	RBTDispose()	SPTDispose()	SKLDispose()
Vložení uzlu	AVLInsert()	RBTInsert()	SPTInsert() SPTInsertBottomUp()	SKLInsert()
Vyhledání uzlu	AVLSearch()	RBTSearch	SPTSearch() SPTSearchBottomUp()	SKLSearch()
Aktualizace uzlu	AVLActualize()	RBTActualize()	SPTActualize() SPTActualizeBottomUp()	SKLActualize()
Vymazání uzlu	AVLDelete()	RBTDelete()	SPTDelete() SPTDeleteBottomUp()	SKLDelete()
Výpis struktury	AVLPrint()	RBTPrint()	SPTPrint()	SKLPrint()

Tabulka 2.1: Přehled operací vytvořené knihovny vyhledávacích metod

	Návrh I 1·malloc()	Návrh II 1·malloc() + 2·memcpy()
1	447	240
2	432	162
3	518	153
4	423	189
5	483	152
průměrně	$460,6 \pm 35,3$	$179,2 \pm 33,2$

Tabulka 2.2: Srovnání způsobů alokace paměti (v počtech tiků)

[18] V tabulce uvedené operace se sufixem BottomUp jsou operace, které vnitřně užívají *bottom-up splay()* operaci (zdola nahoru). Bez tohoto sufixu je užívána *top-down splay()* operace (shora dolů), která by měla být užívána pro praktické nasazení z důvodu vyšší efektivity (má menší paměťové a hlavně časové nároky, jak vyplyne z následující kapitoly).

Kapitola 3

Hodnocení vyhledávacích metod

Při existenci různých vyhledávacích metod, které jsou založeny na různých algoritmech řešení, vzniká požadavek na jejich vzájemné srovnávání, které by ukázalo, k čemu se daná datová struktura hodí, jaké má přednosti, a k čemu je nevhodná. Je zřejmé, že těchto kritérií, dle kterých lze srovnávat, je velké množství.

3.1 Složitost algoritmu

V hodnocení vyhledávacích metod hraje základní roli složitost algoritmu[1]. Touto složitostí můžeme myslet dobu provádění daného algoritmu (časovou složitost), rozsah použité operační paměti (prostorovou složitost) nebo např. množství vzájemně vyměněných dat mezi určitými entitami (tzv. komunikační složitost). Při hodnocení nás zajímá tzv. horní, dolní či průměrný odhad složitosti, neboli maximální, minimální či průměrná složitost algoritmu. U algoritmů jako takových se především hodnotí jejich paměťová a časová složitost. Snahou efektivních datových struktur je oba nároky minimalizovat, i když se často dochází k tomu, že minimalizací jednoho nároku (např. časové složitosti) se zvyšuje jiný nárok (např. prostorová složitost)[2].

Jak plyne ze zadání, práce se zabývá čtyřmi běžnými vyhledávacími metodami, každá se svojí vlastní datovou strukturou. Jsou jimi:

- AVL strom (AVL tree)
- Červeno-černý strom (red-black tree)
- Rozvinutý strom (splay tree)
- Přeskakující seznam (skip list)

Obecně lze k nim říci, že (až na odlišnosti konstant) mají všechny logaritmickou časovou složitost (při n uzlech, je výsledek operace dostupný nejpozději za $\log_2 n$) a lineární paměťovou složitost (při n uzlech je potřeba n prostorových jednotek uzlu), což lze ve vyhledávacích algoritmech považovat za ideální stav. K zachování správných vlastností algoritmu však přispívá i správná programovací technika, která vždy může negativně ovlivnit celý algoritmus.

[1] *Algoritmus je konečná uspořádaná množina úplně definovaných pravidel pro vyřešení nějakého problému.* Je to přesně definovaná konečná posloupnost kroků (příkazů), jejichž prováděním pro každé přípustné vstupní hodnoty získáme po konečném počtu kroků odpovídající hodnoty výstupní. Méně formálně lze pojem algoritmus vysvětlit jako postup, který nás dovede k řešení úlohy. (Zapsáno dle [9]).

[2] Více k tématu složitosti lze např. najít v [16, kapitola 3] a [3, kapitola 2].

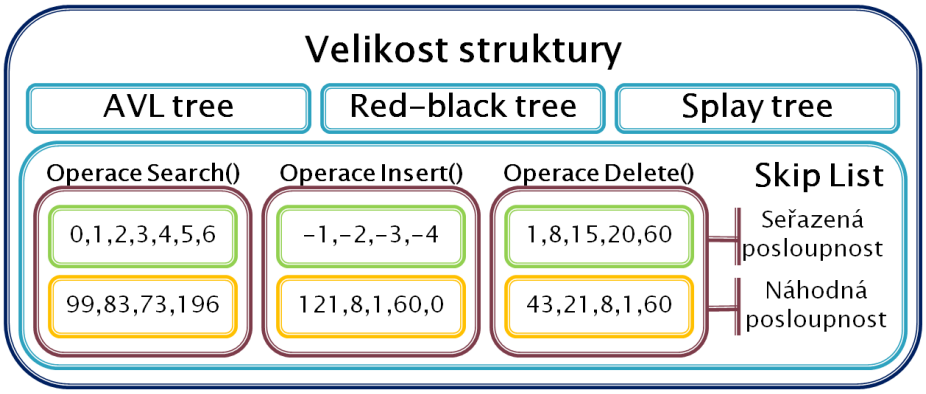
Příkladem může být volba datových struktur, která při nevhodném návrhu může degradovat nejen paměťovou (prostorovou), ale i časovou složitost (přístup k datům).

3.2 Návrh metody hodnocení

Nyní, když jsme úspěšně implementovali zadané vyhledávací metody, přichází na řadu vyhodnocování úspěšnosti jejich implementace a doporučení k jejich užití. Zajímá nás jejich efektivita, což je obecně poměr přínosu (co člověk žádá) na náklady (co provedení operace vyžaduje). Pod náklady je však zahrnut nejeden atribut (až nekonečný počet), takže z těchto principů by nebylo možné pak vytvořit závěry k jejich užití. Proto je nutné se omezit jen na jisté oblasti, které nás budou zajímat.

Z dříve uvedeného nám vyplývá požadavek měřit chování algoritmu na potřebné místo a čas. Co se týče místa, lze napříč spektrem testovaných implementací algoritmů očekávat vždy monotónní výsledek při stejném počtu požadavků odlišujících se v absolutních číslech díky různé velikosti uzlu dané datové struktury, neboť prostorová náročnost na konkrétních uzlech zcela nezávisí. Proto od původního záměru sledovat tento údaj bylo upuštěno. Co se však týče času trvání prováděné operace, který závisí na *vyhledávací metodě*, její příslušné *operaci* a *vyhledávaném klíči*, lze očekávat vzájemně se lišící výsledky. Vyhledávací metody jsou už v práci zadány. Pro hodnocení je však potřeba určit i sledovanou množinu operací nad jistou strukturou konkrétní metody a sledovaný způsob simulace příchodu vyhledávacích klíčů, kterou bude daná operace zpracovávat, protože na jediné hodnotě nemá příliš význam cokoli sledovat. Množinu sledovaných operací je poměrně snadné určit. Nechť jsou jimi operace *vkládání* (*insert*), *hledání* (*search*) a *mazání* (*delete*), protože jsou to operace prakticky nejpoužívanější a nezbytné v každém algoritmu vyhledávací metody. Simulace příchodu hodnot vyhledávacích klíčů již tak snadné není určit. Lze totiž očekávat, že bude-li se jednat o seřazené posloupnosti klíčů, odezva jednotlivých operací napříč spektrem testovaných vyhledávacích metod bude jiná nežli u náhodné posloupnosti. Aby proto nedošlo ke zkreslení výsledků, budeme testovat každou operaci s klíči utvořenými buď na základě *posloupnosti čísel*, nebo tvořenými *náhodně* (s rovnoměrným rozložením) za pomoci pseudogenerátoru, což jsou právě dva extrémy, s jakým můžou klíče přicházet. Každou operaci každé vyhledávací metody budeme tedy několikrát testovat s klíči principiálně vytvořeného dle uvedeného s tím, že se bude jednat v každém běhu testu vždy o novou disjunktní sadu klíčů, protože právě na různých klíčích má význam testovat. Popsané parametry testování nám znázorňuje obrázek 3.1 (str. 45).

Získané množství hodnot trvání jednotlivých operací (s jistými počátečními podmínkami) statisticky zpracujeme. Výsledky pak přehledně uvedeme do tabulek či grafů pro každou vyhledávací metodu, její operaci a použitý způsob generování klíčů. Neměli bychom zapomenout ani na otestování vlivu velikosti dané struktury, abychom ověřili rozsah platnosti závěrů vytvořených na základě výsledků z běhů jednotlivých testů. Vytvořené hodnocení vyhledávacích algoritmů se bude tedy opírat o časové hodnoty získané testováním, které by měly odpovídat teoretickým předpokladům již dříve nastíněných v jednotlivých vyhledávacích metodách v předchozí kapitole. Obdržené experimentální hodnoty nám budou charakterizovat průměrné hodnoty operací jednotlivých vyhledávacích metod. Ty jsou v reálném nasazení nejvýznamnější.



Obrázek 3.1: Parametry testování

3.3 Způsob realizace měření

Změřit trvání prováděné operace je však složitější, než by se mohlo zdát. Již na úplném začátku měření narazíme na potíž jeho prováděné přesnosti, pro kterou vyžadujeme co nejvyšší časovou rozlišitelnost. Ta by měla být nejméně kolem jednotek μs[3], protože vůči počítačům z dřívějších let (doba objevení příslušných metod) došlo k výraznému nárůstu rychlosti, přičemž principy algoritmů zůstaly zachovány. Proto se seznamme s čítači a časovači, které se na počítači nacházejí.

3.3.1 Čítače a časovače

Na počítači lze nalézt 3 typy časovačů, které jsou obvykle dostupné[4]. Jsou jimi:

- „čítač tiků“ – tik counter
- „výkonnostní čítač“ – performance counter
- „čítač taktů procesoru“ – time stamp counter (TSC)

Každý z těchto čítačů má jisté výhody a nevýhody.

Čítač tiků vrací počet ms, které uběhly od doby, kdy byl počítač zapnut. Je řízen příslušným přerušením v ISA sběrnici (spouštěné hranou), které je dnes pravděpodobně virtualizováno na základové desce. Jeho rozlišitelnost je pouze v řádech ms. Přečtení hodnoty čítače vyžaduje sběrnicový přenos. Je-li na sběrnici velké zatížení, dojde k znepřesnění měření (zpomalení) tím více, čím je sběrnice více vytížena. Tento čítač je užíván funkcemi jako timeGetTime() nebo getTickCount().

Výkonnostní čítač pracuje s přesností na μs. Jeho výhodou je přes 1 milion tiků za sekundu[5]. Tento čítač je pravděpodobně odvozen z kontroléru PCI sběrnice. Přečtení čítače opět vyžaduje sběrnicový přenos, jehož zpoždění lze řádově srovnat s předchozím čítačem. Pokud je sběrnice velmi vytížena, na mnoha čipech může časovač skočit o celých 1 až 4

[3] Vycházím z [12, tabulka 2], kde údaje jsou vztaženy na procesorový čas počítače Sun-3/60.
[4] Časovače zpracovány dle [11].
[5] Některé typy sběrnic mají dokonce přes 3 miliony tiků za sekundu.

sekundy(!) vpřed (údajně problém uchovávání 64-bitové hodnoty ve dvou 32-bitových registrech). Tento čítač může být užíván funkcí jako QueryPerformanceCounter(), ale dle vyjádření Microsoftu až jako poslední (třetí) možnost. Předtím se tato funkce snaží získat hodnotu z hardwarového výkonnostního čítače north-bridge (první možnost) nebo čítače taktu procesoru (druhá možnost)[6].

Čítač taktů procesoru TSC je vnitřní 64-bitový registr CPU, který počítá počet taktů (tiků) procesoru od posledního zapnutí či resetu procesoru. Přečtení této hodnoty je tedy velice rychlé. Poznamenejme však, že na rozdíl od předchozích čítačů, které vracely hodnotu v časových jednotkách, hodnota tohoto čítače je vyjádřena v uplynulých taktech procesoru.

Jak z tohoto stručného přehledu plyne, čítač taktů procesoru (TSC) se přesně hodí k zjištění, jak dlouho určitý kus kódu trvá. Přečteme jeho hodnotu, kterou si uchováme. Provedeme testovanou operaci a znovu zjistíme hodnotu čítače. Rozdíl zjištěných hodnot po a před měřením je počet tiků, které uběhly během měření. Vyvstává však otázka, jak získat tuto hodnotu z 64-bitového čítače? Nejjednodušší způsob je za pomoci instrukce asembleru RDTSC (viz [23]), která již dávno byla přidána do instrukční sady procesorů Intel, a proto je dostupná na všech procesorech, které se k této instrukční sadě hlásí nebo tvrdí, že jsou s ní kompatibilní. Uvědomme si, že tento čítač čítá neustále. Můžeme ho však považovat za dostatečně veliký natolik, že v průběhu chodu počítače nikdy nedojde k jeho přetečení. Ověřme:

> Uvažujme (pro dnešní dobu asi nadčasový) procesor o pracovní frekvenci 4 GHz. Aby čítač přetekl, musel by se zvýšit 2^{64} krát (je to 64-bitový čítač). Doba přetečení tohoto čítače tedy činí $T = 2^{64} \cdot \frac{1}{4.10^9} \doteq 4,6.10^9$sekund, což je asi 146 let, neboli doba, po kterou nejel nikdy žádný počítač, natož s podmínkou doby bez jediného přerušení chodu procesoru.

Získanou hodnotu uložíme do `unsigned long long int` (poměrně nový datový typ přidaný v normě C99), což je typ, který je schopen takovouto hodnotu pojmout (bývá to 64-bitový datový typ).

3.3.2 Principy ovlivňující přesnost měření

Během realizace co nejpřesnějšího měření se však musíme nějakým způsobem vyrovnat s principy, které nám způsobují potíže v jejich realizaci, s principy, které nám nějakým způsobem získávané hodnoty ovlivňují. Při tvorbě testovacího programu tedy musíme uvážit:

- škálování frekvence procesoru
- přepnutí kontextu (context switch)
- výpadek stránek (page fault)
- ukládání do pomocných pamětí (caching)

Budeme-li požadovat výsledek měření testované operace v časových jednotkách, musíme zjistit frekvenci, na které CPU běží. Ovšem v dnešních počítačích je třeba vyřešit otázku dynamického škálování frekvence procesoru. Je zcela běžné např. u notebooků, že užívají

[6] Blíže k QueryPerformanceCounter() např. na [13] nebo na stránkách MSDN (Microsoft Developer Network).

podtaktovávání procesoru (neboli snížení frekvence) za účelem snížení spotřeby energie. Tento jev se dostal i k dnešním pracovním stanicím. Protože přepočet na časové jednotky by do procesu měření zákonitě vkládal další nepřesnosti (především z důvodu nepřesného zjištění průběžně se měnící frekvence), proveďme modifikaci dříve uvedeného požadavku (zisk časových hodnot trvání testovaných operací) a srovnávejme dobu trvání testované operace na základě počtu tiků, protože tento čítač tiká vždy podle aktuální frekvence. Navíc díky tomu vyloučíme z naměřených výsledků vliv frekvence procesoru.

Při užívání TSC je třeba vzít na vědomí, že při měření za pomoci tohoto čítače se nezohledňuje multitaskingové prostředí, které v dnešní operačních systémech osobních počítačů typicky užívá preemptivní plánování. Může dojít během měření k nežádoucímu přepnutí kontextu. Protože čítač čítá neustále, započítali bychom do doby trvání měřené operace nejen nejméně dvakrát dobu režie přepnutí kontextu (stovky až tisíce instrukcí), ale i dobu provádění jiných programů, na které bylo přepnuto, než byl procesor navrácen programu, který jsme vytvořili pro účel tohoto měření. Úplné ošetření tohoto principu na současných strojích (operačních systémech) je však nesplnitelné. K přepnutí kontextu totiž může dojít v multitaskingovém prostředí, při ošetření přerušení nebo při přepnutí mezi uživatelským a jaderným módem (záleží na operačním systému)[7]. Toto přepínání kontextu vzniká náhodně, přičemž nelze předem predikovat jeho rozložení. Proto tento princip ošetříme počtem opakování *téhož* testu. Výsledky z těchto stejných měření (mají stejný vstup klíčů) mezi sebou porovnáme a vezmeme z nich nejmenší hodnotu, protože je jisté, že tentýž program bude zpracován vždy stejně rychle a nikdy ne rychleji, nedojde-li ke zdržení zpracování vyvolané okolím.

Výpadek stránek je jev, ke kterému taktéž dochází. Nejkritičtější oblastí na výpadek stránky je dynamická alokace paměti, protože ta je získávána za chodu programu. Pokud program běží, byl již do paměti nahrán. Nedojde-li k pozastavení programu, nehrozí, že by byl z paměti uvolněn. Souvislost s přepínáním kontextu je na místě. Dynamicky alokovaná paměť se však získává až za chodu programu. Zde výpadky stránek jsou zcela běžné, protože stránky jsou zaváděny do paměti jen tehdy, až jsou zapotřebí. Jedním z triviálních řešení by bylo do získané paměti něco zapsat, čímž bychom zajistili jejich potřebu. Program by tak měl při svém nepřerušeném chodu pro sebe zajištěnu dále bezvýpadkovost alokované paměti. Vhodnějším řešením by však bylo alokovanou paměť uzamčít za pomoci funkce specifické pro daný operační systém, např. za pomoci `mlock()`. Pak by bylo zaručeno, že tuto paměť nebude možné odložit. Bohužel však testovací program užívá hodnocení jednotlivých operací knihovny, které zapouzdřují veškeré získávání paměti a činí tak uvedené způsoby nepoužitelnými. Navíc bychom při jejich aplikaci do knihovny uměle zvyšovali režii testované operace, přičemž tyto návrhy děláme proto, aby k nečekanému zvyšování režie operace znepřesňující měření nedocházelo. Proto se omezme jen na zjišťování nejmenší hodnoty průběžného čítače jako u přepínání kontextu a spolehněme na to, že dostatečný počet téhož testu tyto nepřesnosti odstraní. Naštěstí tento problém dnes nebývá příliš palčivý, protože dnes většinou bývá dostatek volné paměti i paměti na to, aby program po jistou dobu nebyl hned (dočasně) odkládán na disk, byl-li na nepříliš dlouhou dobu přerušen.

Narozdíl od přepínání kontextu a výpadku stránek, které výsledek měření podhodnocují, ukládání do pomocných pamětí výsledek měření nadhodnocuje. I to je paradoxně chybně. Pro bližší pochopení nejdříve zjistěme velikost struktury dat o 1000 uzlech pro jednotlivé metody:

Uvažujme, že velikost typu `int` a velikost kteréhokoliv ukazatele (např. `void *`)

[7] Blíže k přepnutí kontextu se lze dočíst v [19].

je právě 4 byte[8] (dále nazýváno jako položka). Dále pro jednoduchost uvažujme, že typ klíče je `int` a velikost datové struktury každé metody je 0 (nemá žádné členy) a paměťové uspořádání uzlu je dle obrázku 2.17(b), str. 41 (tak je alokace nové položky v knihovně každé struktury vyhledávací metody vždy implementována). Zarovnání paměti zanedbejme. Podíváme-li se do hlavičkových souborů jednotlivých struktur uzlů, zjistíme, že AVL uzel má 6 položek, RBT uzel 7 položek, SPT uzel 5 položek a SKL prvek 14 položek při volbě maximální úrovně 10. Výsledná velikost struktury S (o 1000 prvcích) v dynamicky alokované paměti na hromadě tedy činí:

Výpočet struktury [byte]	Velikost [kB[9]]
$S_{AVL} = 1000 \cdot 6 \cdot 4$	$23,4$
$S_{RBT} = 1000 \cdot 7 \cdot 4$	$27,3$
$S_{SPL} = 1000 \cdot 5 \cdot 4$	$19,5$
$S_{SKL} = 1000 \cdot 14 \cdot 4$	$54,7$

Vezmeme-li v úvahu, že velikost L1 cache procesoru je 32 KiB (Intel Pentium III), přičemž dle [22] je její užívání rozděleno na 16 + 16 KiB pro data a instrukce, vidíme jasně, že tyto velikosti struktur se do L1 cache nevejdou. Pokud budeme jednotlivé metody srovnávat na typ posloupnosti vstupních klíčů (uspořádaná, náhodná), nesmíme zapomenout vzít v úvahu zkreslující vliv ukládání do pomocných pamětí (caching), protože nelze najít způsob, jak tomuto jevu zabránit.

3.3.3 Testovací program

Program pro své spuštění požaduje čtyři argumenty. Jsou jimi:

1. počet klíčů, které vložit do struktury, než ji začneme testovat
2. počet klíčů, na kterých se testovaná operace měří
3. počet testů, které provést nad nově vygenerovanou sadou vzájemně disjunktních klíčů
4. počet opakování každého měření v každé metodě (při identické sadě klíčů)

Ačkoli by se mohlo zdát, že testovací program pro svůj chod potřebuje zbytečné množství argumentů, není tomu tak. Vysvětleme si tedy jejich význam k testování.

ad 1 Tato položka ovlivňuje vzájemnou rozlišitelnost vypovídacích hodnot operací nad jednotlivými strukturami metod. Pokud by byla počáteční (startovací) struktura příliš malá (jednotky až desítky položek), z výsledků by nebylo možné vyvodit jakékoli vzájemné srovnání, protože kterákoli testovaná operace by se jevila obdobně rychlá. Naštěstí všechny testované vyhledávací metody jsou primárně zaměřeny na vyhledávání ve velkém množství. Nehrozí tak riziko, že bychom při vzájemném srovnávání při nevhodném (příliš nízkém) zadání této položky dospěli k mylnému závěru, že jistá metoda je rychlejší, ačkoli by skutečnost při jiném počtu prvků byla jiná.

[8] Skutečnou hodnotu na dané architektuře lze snadno zjistit pomocí klíčového slova `sizeof()`.

[9] Striktně vzato, správné označení jednotky by mělo být KiB, protože správně je 1 KiB = 1024 byte a 1 kB = 1000 byte, v praxi se však označení KiB neuchytilo a místo něj se v informačních technologiích užívá stále užívá kB ve významu 1 kB = 1024 byte.

ad 2 Tato položka byla přidána do testování pro získání výsledků charakteristických pro průměrnou dobu trvání testované operace. Filozofie získání průměrné hodnoty trvání operace pro konkrétní počáteční stav struktury je založena na tom, že operace se provede tolikrát, kolik je tato hodnota, neboli kolik dostane různých klíčů. Celkovou hodnotu pak vydělíme počtem tohoto opakování, čímž získáme průměrnou dobu trvání testované operace pro jistou počáteční strukturu.

ad 3 Tato hodnota v podstatě znamená počet vygenerování nové sady vzájemně disjunktních klíčů[10]. Počet těchto klíčů je dán součtem hodnot z 1 a 2. Pro testování byla zvolena disjunktní sada klíčů z důvodu testování jen „plnokrevných" operací, protože lze očekávat, že tak v praxi bude operace nejvíce používána. Asi málokdo bude chtít odstraňovat něco, když předtím to tam nevložil, nebo vkládat něco, když už to tam je, i když samozřejmě vytvořená knihovna vyhledávacích metod i s těmito žádostmi počítá. Konkrétní chování každé operace je popsáno v hlavičkových souborech příslušných metod.

ad 4 Tato hodnota udává, kolikrát se mají provést tytéž testy operací jednotlivých struktur nad stejnou sadou vygenerovaných klíčů, neboli kolikrát tentýž test opakovat. Zadává se z důvodu potíží, s kterými se potýkáme v průběhu měření. Slouží nám ke zpřesnění téhož měření tím, že za dobu trvání testované operace se bere vždy nejnižší hodnota získaná z jednotlivých měření. Zdůvodnění považovat vždy nejnižší hodnotu za správnější bylo rozebráno v části 3.3.2 (str. 46).

Výsledkem každé testovací sady klíčů je počet tiků čítače pro každou operaci každé struktury. V závislosti na počtu testů (počtu jistým způsobem vygenerovaných sad klíčů) obdržíme počet výsledků, ze kterých určíme průměrnou hodnotu a směrodatnou odchylku. Tyto hodnoty pak přehledně uvedeme v další kapitole (3.4). Průměrnou hodnotu $\overline{x}$ a směrodatnou odchylku s vypočteme dle $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ a $s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$. Protože však uvedený vztah pro výpočet odchylky by vedl na nezbytný dvojí průchod všemi výsledky (nutnost nejdříve znát průměrnou hodnotu), můžeme si dovolit provést malou optimalizaci. Na základě jednoduchého odvození[11] lze výpočet odchylky převést na $s = \sqrt{\frac{1}{n}(\sum_{i=1}^{n} x_i^2) - \overline{x}^2}$. Pak již nemusíme uchovávat si všechny výsledky např. někde v poli o předem neznámé velikosti a vystačíme si jen s dvěmi proměnnými na jednu měřenou operaci, do kterých si budeme ukládat součet a součet druhých mocnin obdržených výsledků. Mělo by se však upozornit na typ takovýchto proměnných. Ačkoli by bylo nejlepší tyto součty uchovat v přesných číslech s pevnou řádovou řádkou, součet druhých mocnin tuto možnost zcela vylučuje. I když z tohoto důvodu lze začít uvažovat o původním a méně efektivním vztahu pro výpočet odchylky, ani u součtu všech výsledků na tom nebudeme o mnoho lépe. Proto musíme použít aritmetiku s plovoucí řádovou čárkou, kde hlavní nebezpečí tkví v přesnosti jednotlivých datových typů[12]. Jak jsem se prakticky přesvědčil, typ `double` je pro `unsigned long long`

[10] Způsob zajištění vzájemně disjunktních klíčů je na základě vlastního kongruentního generátoru s tím, že stačí jen ověřovat, že vygenerovaná hodnota není rovna první vygenerované hodnotě.

[11] Při úpravách mějme na paměti, že $\overline{x}$ je známá konstanta a platí: $\sum_{i=1}^{n} \overline{x}^2 = n \cdot \overline{x}^2 \quad \frac{1}{n}\sum_{i=1}^{n} x_i = \overline{x}$.
$s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2\overline{x}\cdot x_i + \overline{x}^2)} = \sqrt{\frac{1}{n}(\sum_{i=1}^{n} x_i^2) - \frac{2\overline{x}}{n}(\sum_{i=1}^{n} x_i) + \frac{1}{n}(\sum_{i=1}^{n} \overline{x}^2)} =$
$\sqrt{\frac{1}{n}(\sum_{i=1}^{n} x_i^2) - 2\overline{x}^2 + \overline{x}^2} = \sqrt{\frac{1}{n}(\sum_{i=1}^{n} x_i^2) - \overline{x}^2}$

[12] Potíží v úvahách o přesnosti a rozsahu jednotlivých datových typů je to, že počet bitů pro žádný datový typ v jazyce C není přesně dán, jsou jen známy jejich vzájemné relace tak, že větší datový typ užívá stejně nebo více bitů. Z tohoto důvodu všechny zde uváděné bitové velikosti vychází ze standardu IEEE 754, kde

`int` málo přesný. (Proto o typu `float` nemůže být ani řeč.) A není divu. Je to sice prakticky 64-bitový typ (viz [20]), ale těchto 64 bitů je rozděleno na 52 bitů mantisy, 11 bitů exponentu a 1 bitové znaménko. Nelze tak přesně převést získanou 64-bitovou hodnotu do tohoto typu. Je třeba použít typ `long double`, který je dle [21] nejméně 80-bitový a má vyšší počet bitů věnovaných na mantisu (nejspíše 64 bitů).

Přehledový zápis implementace testovacího programu lze zapsat následovně:

- Získání jednotlivých argumentů programu (označme je např. `treeset`, `testset`, `count`, `repetition`)
- Vytvoření pomocných proměnných pro uchování průběžného součtu výsledků a průběžného součtu druhých mocnin výsledků a jejich inicializace pro každou měřenou operaci každé metody (vkládání, hledání, mazání nad AVL tree, red-black tree, splay tree, skip list)
- ```
  int array[treeset+testset]; // pole sady náhodných čísel
  for(int i=0; i<count; i++){ // počet testů s novou sadou vygenerovaných klíčů
    getKeySet(seed, array, treeset+testset); // vygenerování sady klíčů
    int *const testSet = &treeSet[treeset]; // pole prvků, na kterých testujeme
    unsigned long long start, end; // počáteční hodnota a koncová hodnota čítače
    for(přes jednotlivé měřené operace operace a metody){
      for(int j=0; j<repetition; j++){ // počet opakování téhož testu (zpřesňování)
        /* inicializace a počáteční naplnění testované struktury */
        start = rdtsc(); // zjištění aktuální hodnoty čítače na začátku testování
        /* provedení jisté operace konkrétní metody nad testovanou sadou klíčů */
        end = rdtsc(); // zjištění aktuální hodnoty čítače na konci testování
        if(result > end-start) // uchovávání si vždy menší hodnoty
          result = end-start; // doba trvání
        /* provedení téže operace nad ostatními metodami */
        forTestsDispose(&test); // uvolnění struktury
      }
      /* přičtení (získaný výsledek/testset) do průběžného součtu výsledků
         a součtu druhých mocnin výsledků k dané operaci a struktuře */
    }
  }
  ```
- Zpracování získaných hodnot (výpočet) a výpis průměrné hodnoty a směrodatné odchylky pro každou zúčastněnou operaci a metodu v provedeném testu

3.4 Výsledky měření

Stanovme si směr našeho zjišťování na zodpovězení těchto otázek:

- Jak rychlé jsou jednotlivé operace v různých strukturách?

velikost uvažovaných datových typů používanými na mé pracovní stanici jsem úspěšně konfrontoval s tímto standardem, i když v jiných podmínkách se uváděné velikosti datových typů můžou lišit (standard jazyka to nezaručuje). Důvodem nestanovení přesné velikosti datového typu v jazyce C je zajištění přenositelnosti do různých oblastí, které by nemusely mít možnost pevně zadanou velikost splnit.
  ```

- Jak ovlivňuje uspořádání vstupních klíčů trvání jednotlivých operací?
- Jak závisí trvání jednotlivých operací na velikosti struktury?

Uvažované operace jsou *vkládání* (*insert*), *hledání* (*search*) a *mazání* (*delete*), uvažované vstupy jsou s klíči vygenerovanými *náhodně* (*random*) nebo *uspořádaně* (*sorted*). Z testovaných struktur budeme uvažovat *AVL strom* (dále *AVL*), *červeno-černý strom* (dále *RBT*), *rozvinutý strom s top-down splay* (dále *SPT*), *rozvinutý strom s bottom-up splay* (dále *SPTBU*) a *přeskakující seznam* (dále *SKL*).

Před testováním byl počítač uveden do „klidového stavu“ – byly zastaveny a ukončeny všechny uživatelovy aplikace, vypnuty nedůležité služby systému, odpojena síť, minimalizován počet běžících procesů, zvýšena priorita testovacího programu. Pro zvýšení přehlednosti hodnot výsledků bylo využito techniky podbarvování položek, kde hodnota méně podbarvená (světlejší) vypovídá o méně spotřebovaném výpočetním času procesoru, a lze ji tak považovat za lepší. Všechny hodnoty jsou implicitně uvedeny v počtech tiků procesoru, pokud není uvedeno jinak. Chceme-li získat představu o tom, kolik tato hodnota odpovídá v jednotkách času, stačí tuto hodnotu vydělit frekvencí procesoru. Př.: Frekvence procesoru je 2000 MHz, počet tiků 500, doba trvání tedy je: $t = 500/2000 = 0,25\,\mu$s.

3.4.1 Závislost trvání jednotlivých operací v různých strukturách

Pro měření byla použita počáteční struktura o 3600 položkách. Každá operace zpracovala 500 klíčů. S těmito počty bylo provedeno 200 nezávislých testů o zcela nově vygenerované sadě klíčů určených pro počáteční strom a testování, kde každý test operace byl opakován 50 krát. Maximální úroveň přeskakujícího seznamu byla zvolena 12, protože pak očekávaný počet prvků v této struktuře je 2^{12} (4096), což odpovídá zpracovávanému množství klíčů. Pro vzájemné srovnání stejných operací v různých strukturách bylo provedeno měření nad náhodně vygenerovanou sadou klíčů (s rovnoměrným rozložením) a následně nad uspořádanou sadou klíčů, aby se zjistilo, zda má uspořádanost klíčů při srovnávání stejných operací různých metod nějaký vliv. Výsledky měření vidíme v tabulce 3.1. (Hodnoty jsou uvedeny v počtech tiků procesoru.)

Jak je názorně vidět z této tabulky (3.1), různé vyhledávací metody mají (dle očekávání) různou dobu trvání jednotlivých operací, ale navíc se ukázalo, že tyto doby závisí na uspořádání klíčů. Vyjdeme-li z předpokladu, že vstupní klíče nejsou nijak uspořádány, shledáme, že zatímco AVL je nejlepší pro vyhledávání, na dobu operací vkládání a mazání, které bývají často považovány za režijní, je nejlepší RBT. To je způsobeno tím, že RBT je méně striktní na vyváženost (povoluje až dvakrát delší cestu vůči jiné cestě z téhož uzlu). Je-li však vstup jistým způsobem lokalizován (v našem případě uspořádán), projeví se pozitivní účinek *splay top-down* operace u SPT, která zajišťuje udržování naposledy žádaných uzlů co nejblíže kořenu stromu. Protože se jedná jen o prostý strom, jsou všechny operace rychlejší (ovšem se zdůrazněním předpokladu lokalizace klíčů). Je ale také patrné, že efektivita *splay* operace hraje zásadní vliv. *Splay bottom-up* operace, která vyžaduje průchod zdola nahoru, vkládá vysokou režii, která ve svém důsledku způsobí nejlépe (nejvýše) srovnatelné výsledky s vyvažovanými stromy (AVL a RBT). Protože má vysoké trvání i při příznivém vstupu, v praxi nemá žádného uplatnění a nezbývá než ji odkázat nejvýše do školního prostředí k demonstraci jejího principu, který je skutečně velice jednoduchý. Lze předpokládat, že v konkurenci s lineárními vyhledávacími metodami by se mohla uplatnit, ale v porovnání s logaritmickými metodami nemá své opodstatnění. V tomto hodnocení je také poměrně

(a) Náhodně vygenerovaná posloupnost klíčů

	Insert	Search	Delete
AVL	$863,9 \pm 4,3$	$476,2 \pm 2,1$	$1037,2 \pm 3,7$
RBT	$678,2 \pm 4,4$	$488,1 \pm 2,1$	$599,4 \pm 2,0$
SPT	$990,8 \pm 4,2$	$558,6 \pm 0,1$	$740,0 \pm 3,1$
SPTBU	$2188,0 \pm 6,0$	$1306,5 \pm 0,1$	$1519,4 \pm 11,4$
SKL	$1308,1 \pm 11,7$	$853,6 \pm 10,4$	$1009,6 \pm 18,1$

(b) Uspořádaná posloupnost klíčů

	Insert	Search	Delete
AVL	$568,5 \pm 0,9$	$309,7 \pm 0,2$	$856,7 \pm 0,1$
RBT	$604,3 \pm 0,7$	$349,3 \pm 0,1$	$449,6 \pm 0,5$
SPT	$439,7 \pm 0,7$	$215,2 \pm 0,1$	$216,7 \pm 0,2$
SPTBU	$688,9 \pm 2,1$	$493,5 \pm 1,2$	$528,3 \pm 0,1$
SKL	$1099,1 \pm 26,0$	$622,4 \pm 17,9$	$616,3 \pm 51,4$

Tabulka 3.1: Závislost trvání jednotlivých operací v různých strukturách

špatně na tom i SKL, jehož výsledky nejsou při srovnávání s ostatními logaritmickými vyhledávacími metodami nijak oslnivé.

3.4.2 Vliv uspořádanosti vstupních klíčů v různých strukturách

Pokud se podíváme k zodpovězení této otázky na tabulky 3.1(a) a 3.1(b), a budeme-li je vzájemně porovnávat, zjistíme, že uspořádání výrazně zrychluje dobu zpracování pro všechny struktury. Bohužel tento závěr je mylný, protože při měření doby operace dochází k ukládání do pomocných pamětí dříve projitých položek. U uspořádaných klíčů je toto ukládání (na straně procesoru, L1 cache) vysoce efektivní, protože se neustále prochází téměř stejné uzly, neboli cesta je víceméně stejná. U náhodné posloupnosti toto neplatí, a proto ukládání do pomocných pamětí se tolik neprojeví. Zatímco při vzájemném srovnávání operace v různých strukturách tento jev příliš nevadil (využití pomocných pamětí pro uložení jednotlivých struktur bylo zhruba stejné), při srovnávání téže operace na různé vstupy klíčů nám již toto ukládání vadí. Proto při měření testované operace budeme zpracovávat jen jednu položku, protože tím dosáhneme toho, že při měření doby pro první průchod nebude cesta zcela v pomocných pamětích uložena. Tento počin však s sebou obnáší i to, že každý obdržený výsledek tohoto měření nebude brát na zřetel průměrnou dobu trvání operace nad konkrétní konfigurací struktury, ale jen aktuální dobu trvání nad právě jedním uzlem, takže ve svém principu jeden test nebude příliš odpovídat průměrné době trvání měřené operace. Naštěstí lze předpokládat, že zvýšený počet testů tento nedostatek potlačí s tím, že dojde ke zvýšení odchylky naměřených hodnot.

Pro měření tedy byla použita počáteční struktura o 4090 položkách. Každá operace zpracovala právě 1 klíč. S těmito počty bylo provedeno 400 nezávislých testů o zcela nově vygenerované sadě klíčů určených pro počáteční strom a testování, kde každý test operace byl opakován 50 krát. Maximální úroveň přeskakujícího seznamu byla zvolena 12, protože pak očekávaný počet prvků v této struktuře je 2^{12} (4096), což odpovídá zpracovávanému množství klíčů. Výsledky měření vidíme v tabulce 3.2. (Hodnoty jsou uvedeny v počtech tiků procesoru.)

(a) Operace vkládání (insert)

	AVL	RBT	SPT	SPTBU	SKL
Random	$701,8 \pm 67,8$	$606,5 \pm 14,8$	$465,3 \pm 69,1$	$876,4 \pm 88,7$	$908,6 \pm 55,6$
Sorted	$708,0 \pm 14,6$	$741,9 \pm 2,3$	$340,6 \pm 2,0$	$470,4 \pm 1,8$	$820,7 \pm 32,9$

(b) Operace hledání (search)

	AVL	RBT	SPT	SPTBU	SKL
Random	$288,9 \pm 32,2$	$298,9 \pm 33,8$	~~$98,6 \pm 9,6$~~	~~$240,3 \pm 17,9$~~	$452,6 \pm 51,2$
Sorted	$293,2 \pm 10,5$	$481,7 \pm 3,8$	~~$93,1 \pm 1,2$~~	~~$239,1 \pm 2,8$~~	$380,3 \pm 77,7$

(c) Operace mazání (delete)

	AVL	RBT	SPT	SPTBU	SKL
Random	$794,5 \pm 29,8$	$400,3 \pm 17,1$	~~$192,9 \pm 12,8$~~	~~$314,7 \pm 13,6$~~	$504,5 \pm 48,2$
Sorted	$1266,9 \pm 26,2$	$555,1 \pm 1,9$	~~$185,8 \pm 5,0$~~	~~$326,0 \pm 5,0$~~	$485,6 \pm 70,3$

Tabulka 3.2: Vliv uspořádanosti vstupních klíčů v různých strukturách

Z výsledků měření vyplývá, že různé struktury a jejich operace jsou různě citlivé na uspořádanost klíčů. O AVL můžeme tvrdit, že není příliš citlivý na uspořádání klíčů, i když z principu stromu je zaměřen na náhodný vstup. To doslova platí pro operaci vkládání a vyhledávání, kde, pokud vezmeme v úvahu odchylku měření, jsme získali zcela stejné výsledky. Výhodou totiž u vkládání, které musí řešit vyvažování, je, že provede nejvýše jednu rotaci. Jediným slabým místem je operace mazání, která při uspořádaném vstupu je pomalejší. To je způsobeno tím, že dochází k vyvažování stromu od odebraného uzlu až ke kořenu stromu, neboli nastává často nejhorší možný případ v operaci mazání AVL stromu. RBT je také stromová struktura, ale na rozdíl od AVL se nám tu už projevuje jisté zpomalení v naprosto všech operacích při uspořádaném vstupu. Je to hlavně dáno tím, že u seřazeného vstupu vždy dochází k více úpravám než u náhodného vstupu, a proto je vkládání i mazání pomalejší. Zpomalení u vyhledávání lze zdůvodnit tím, že strom bude mít skutečně nejhorší vyvažovanou konfiguraci (dvakrát delší cesta než jiná). Na základě výsledků u SPT a SPTBU při uvážení odchylky měření lze dojít k závěru, že operace vyhledávání a mazání na rozdíl od vkládání nezávisí na uspořádanosti. Ovšem musíme vzít v úvahu to, že k otestování úspěšného hledání či mazání je třeba mít nějaký počáteční strom. Ten je tvořen vložením odpovídajícího počtu prvků. Zároveň však při každém vkládání dochází k umístění vkládaného uzlu vždy do kořene stromu (zajišťuje tzv. splay operace stromu). Testovací program je záměrně tvořen tak, aby vyhledával nebo mazal zadaný počet prvků, se kterými se naposledy pracovalo, protože tímto se může tento vstup považovat i za jistou lokalizaci výběru klíčů. Jenže pokud zadáme měřit trvání operace nad právě jedním klíčem (zdůvodněno výše), budeme bez ohledu na vstup právě u operace hledání nebo mazání testovat dobu provedení činnosti jen nad kořenovým uzlem stromu a různé uspořádání vstupu klíčů se nám neprojeví. Proto tyto hodnoty nelze pro hodnocení akceptovat. Naštěstí popsaný problém se u operace vkládání neprojevuje, protože se uzel přidává do jistého uspořádání stromu, které na uspořádání klíčů silně závisí. Z testování vidíme, že uspořádání (a potažmo tedy lokalizace klíčů) příznivě ovlivňuje trvání operace. Proto na základě platnosti tohoto závěru nad operací vkládání a dále s využitím výsledků z 3.1(a) a 3.1(b) u SPT a SPTBU můžeme tvrdit, že všechny operace vykazují zkrácenou dobu trvání při uspořádání (lokalizaci) klíčů.

Poslední srovnávanou strukturou je SKL. Vezmeme-li v úvahu odchylky měření, dospějeme jednoznačně k hodnocení, že SKL zcela nezávisí na vstupu klíčů, a to v naprosto všech uvažovaných operacích bez výjimky.

3.4.3 Závislost trvání jednotlivých operací na velikosti struktury

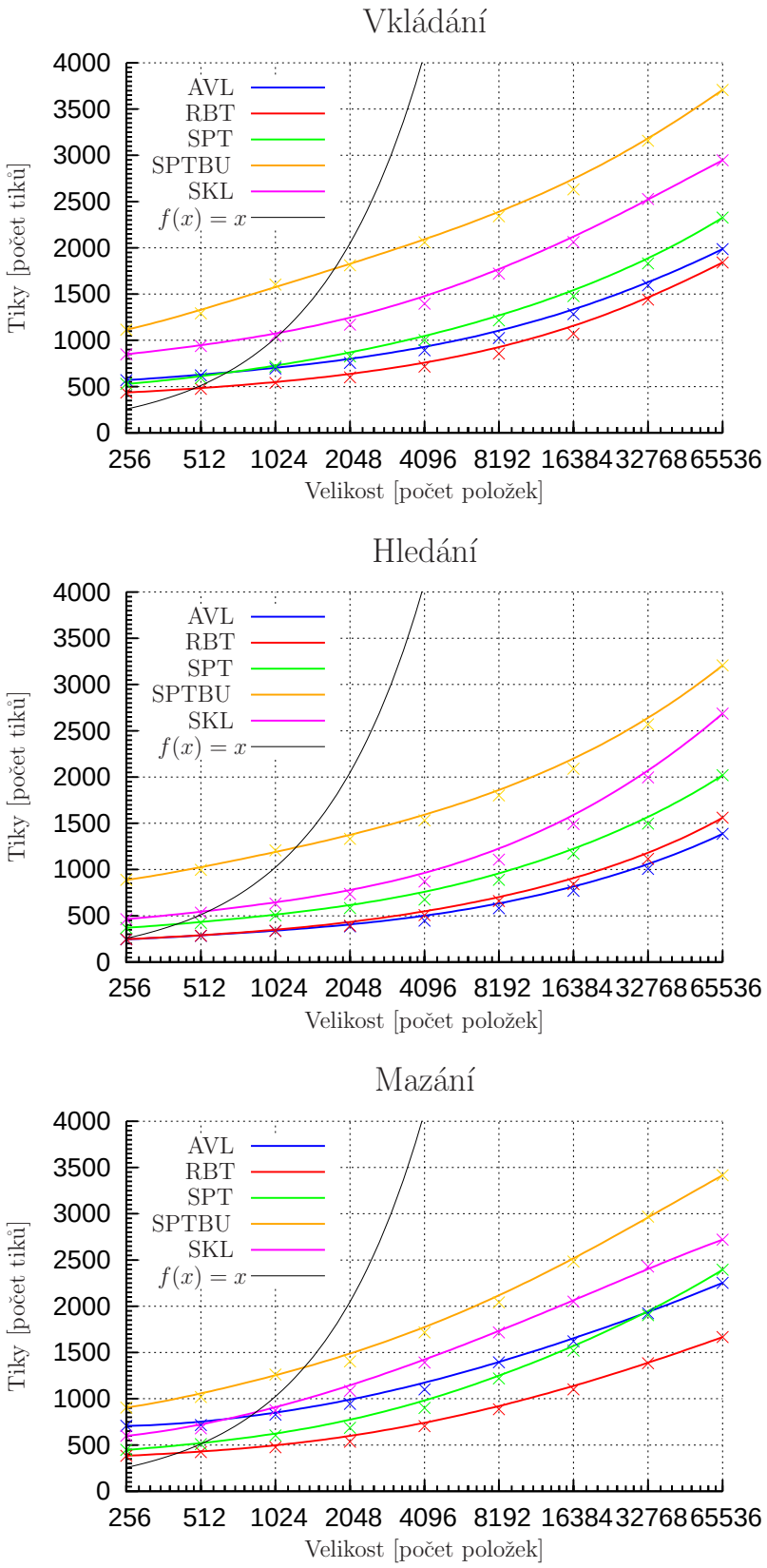
Dosud všechny závěry byly vztaženy na vyhledávací metody, kde velikost jejich struktury byla kolem 4096 položek. Přesvědčme se však o tom, zda uvedené výsledky lze zobecnit na struktury, které budou mít jiný počet položek. Testy probíhaly vždy nad příslušným počtem položek vyznačenými v grafu, přičemž testovaná operace zpracovávala vždy polovinu vygenerovaných testovacích klíčů, abychom při hodnocení zachovali co největší objektivnost[13]. Dodejme, že při testování na velikost byla u přeskakujícího seznamu vždy správně volena jeho odpovídající úroveň, jak již dříve bylo rozebráno v části 2.1.4 (str. 24). Výsledky měření jsou přehledně zaneseny v grafech v 3.3 (náhodný vstup klíčů) a 3.4 (uspořádaný vstup klíčů).

Podíváme-li se na jednotlivé průběhy grafů, zjistíme, že jsme tímto prakticky prokázali, že vytvořené závěry z předchozích měření v tabulce 3.1 jsou v diskutovaných vyhledávacích metodách nezávislé na množství položek, se kterými daná metoda pracuje.

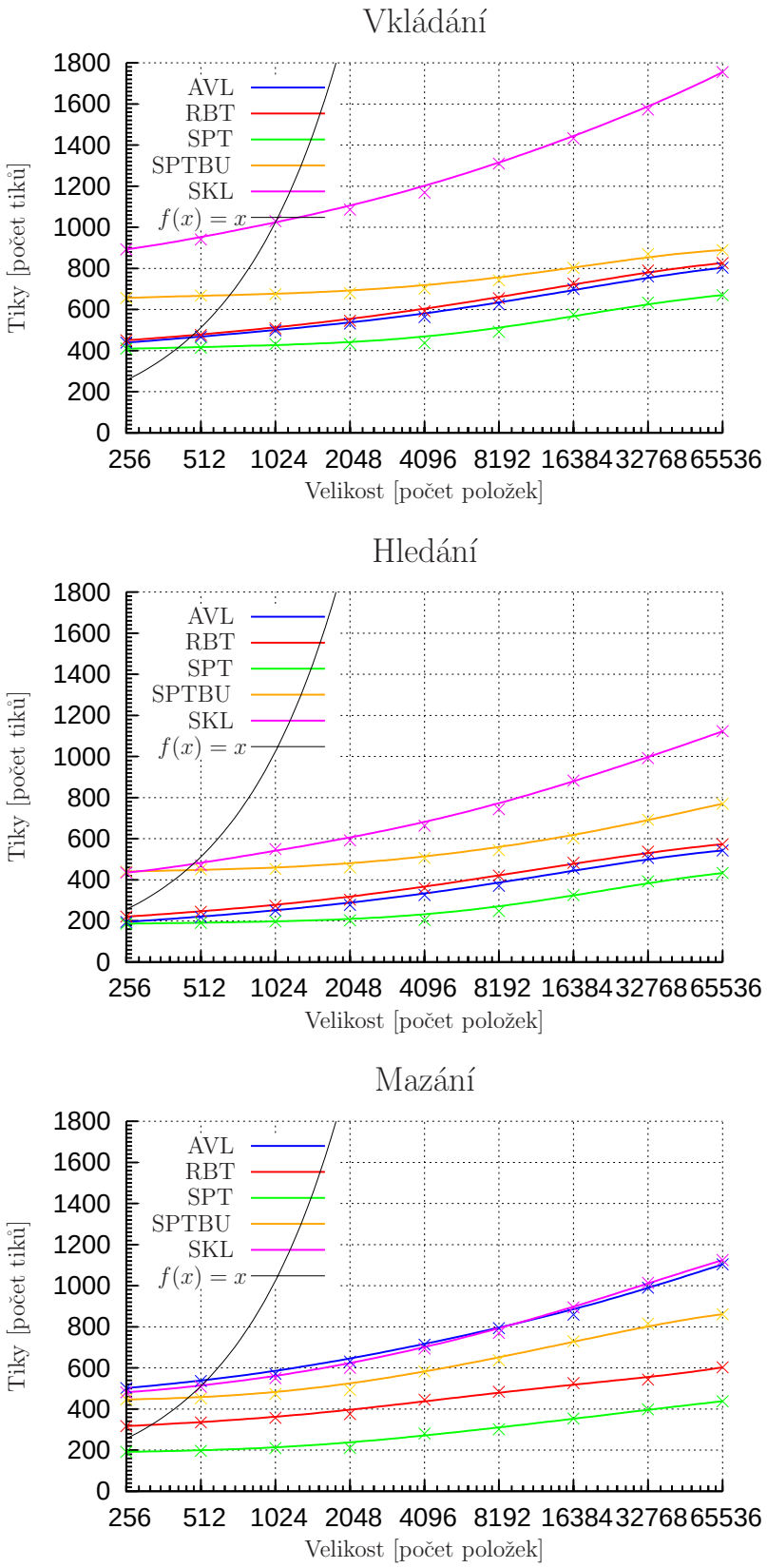
Rozeberme si blíže jednotlivé grafy. Na úvod vezměme v potaz, že osa x má logaritmické měřítko a dále že srovnávání téže operace při různém vstupu klíčů (srovnávání výsledků 3.3 a 3.4) je neadekvátní opět díky používání pomocných pamětích. Můžeme však mezi sebou srovnávat jednotlivé metody i operace v rámci stejného vstupu při různém počtu hodnot.

Proveďme nejdříve doplnění na srovnání mezi jednotlivými operacemi. Budeme-li uvažovat náhodný vstup klíčů, zjistíme, že AVL je skutečně nejrychlejší metoda pro vyhledávání, i když poměrně těsně s ní soupeří metoda RBT. U vkládání a hlavně mazání je RBT oproti AVL výhodnější. Tyto uvedené vztahy vyplývají především z požadavků na vyvažování, které ovlivňují počet rotací, jejichž maximální počet byl již diskutován u jednotlivých metod (u AVL vkládání max. 1 rotace, mazání max. rotace dány počtem uzlů na cestě ke smazanému uzlu stromu, RBT vkládání max. 2 rotace, mazání max. 3 rotace, nezaměňovat rotace s pouhým přebarvováním). Proto skutečnost, že trvání operace mazání je u AVL vysoká nás nemůže překvapit. Obdobné platí i při uspořádání klíčů. Vysvětlení, proč si při vkládání AVL a RBT svoji pozici vyměnily je právě v nepříznivém vstupu klíčů pro tyto metody, kdy se u RBT projevila nutnost zvýšeného počtu vyvažování než u AVL. Zaměříme-li pozornost na srovnávání SPT vůči AVL, shledáme, že při náhodném vstupu je SPT pomalejší, nejedná-li se o nízký počet uzlů, jejichž hranice je pro jednotlivé operace stanovena různě. Při uspořádaném vstupu, který nám představuje opačný extrém lokalizace klíčů je však SPT nejvýhodnější vyhledávací metodou vůbec. Záleží však vysoce na efektivní implementaci *splay* operace, která vždy přemisťuje naposledy žádaný uzel do kořene stromu. Zatím co *top-down splay* operace užívaná v SPT modifikuje a přemísťuje uzly hned na jeden průchod shora dolů, metoda označená SPTBU s *bottom-up splay* operací, která nejdříve

[13] Zvolili-li bychom pevný počet klíčů, které testovaná operace bude zpracovávat, a měnili-li bychom při různém spuštění testování počet položek, které budeme do dané struktury ukládat (právě náš případ), měnila by se vypovídací hodnota obdrženého výsledku při každém novém spuštění testu. Získaný výsledek, který při malém počtu položek lze do jisté míry považovat za průměrnou hodnotu jisté metody, by při zvyšování velikosti struktury svůj průměrný charakter ztrácel a přecházel ke konkrétnější hodnotě závislejší nad aktuálním uspořádáním. To by však vedlo k chybným závěrům. Jako příklad takového nesprávného závěru přesně plynoucího ze zanedbání tohoto jevu je, že vyhledávání při vyšším počtu položek i při náhodném vstupu dat metodou SPT je rychlejší, než u AVL. Ve skutečnosti díky takto chybné volbě argumentů testovacího programu by docházelo jen k lokalizaci klíčů, nad kterými je SPT opravdu rychlejší než AVL.



Tabulka 3.3: Závislost trvání jednotlivých operací na velikosti struktury *při náhodném vstupu klíčů*



Tabulka 3.4: Závislost trvání jednotlivých operací na velikosti struktury *při uspořádaném vstupu klíčů*

vyžaduje znát cestu k přemisťovanému uzlu (dvouprůchodový způsob tam a zpět), se neosvědčila. Medodu SKL lze obecně považovat při srovnávání se stromovými strukturami za pomalou. Ukazuje se, že režie tvorby hierarchie odkazů je výpočetně (a tedy časově a bohužel zde i paměťově) náročnější, než průběžné vyvažování stromu.

Obrátíme-li pozornost k hlavní otázce této části, jak se chovají logaritmické vyhledávací metody v závislosti na velikosti, zarazíme se hned, že nemají přesný logaritmický průběh, nýbrž něco mezi ideálním průběhem logaritmické a lineární funkce. Logaritmická funkce na zlogaritmované ose x je přímka rovnoběžná s touto osou, lineární funkce $f(x) = x$ byla pro srovnávání do grafu vyznačena. Očekávaný průběh logaritmických vyhledávacích metod by tedy měl být prostá přímka (pokud máme v ose x logaritmické měřítko o stejném základu jako logaritmická funkce) při různém umístění do počáteční hodnoty v ose y (různé konstanty) a (vzestupného) sklonu (multiplikativní konstanty logaritmické funkce). To se však neděje z důvodu negativních vlivů, které tento platný předpoklad zkreslují. Těch je veliké množství. Za hlavní aktéry lze považovat vliv ukládání do pomocných pamětí.

Nejzajímavějšími údaji, které lze v chování zkoumaných metod v závislosti na velikosti (počtu položek) pozorovat, jsou body protínání jednotlivých funkcí (operací příslušných metod). Lze tak názorně dokumentovat, kdy principiální jednoduchost operace jisté metody je či není vhodnější jak tatáž operace pokročilejší metody. Jak je vidět, jedná se o otázku vhodnosti použití jednoduchých metod SPT a SKL vůči AVL. Obecně jsou to však jen výjimky, které v rámci celé metody při stejném rozložení vstupu klíčů, pro níž budeme požadovat kompletní základní sadu operací (vkládání, hledání, mazání), nemají svůj význam ke zdůvodnění oprávněnosti jejich upřednostnění.

Úmyslně vynesený graf nejjednodušší lineární funkce $f(x) = x$ nám nám přehledně dokumentuje význam logaritmických vyhledávacích metod a výhodnost jejich použití. Z průniků jednotlivých metod s touto nejjednodušší lineární funkcí lze usuzovat, že logaritmické metody je zcela jednoznačně vhodné nasadit od 1024 (zhruba 1000) a více položek. Rozmezí 512 až 1024 patří mezi přelomové, kdy lze diskutovat o významu nasazení těchto metod. Vzhledem však k tomu, že vzorová lineární funkce je ta nejjednodušší, kde průběh skutečných lineárních vyhledávacích metod (typicky vyhledávání nad seznamy) bude prakticky horší než zanesená funkce, přiklonil bych se k užívání logaritmických metod. Zhruba do 500 položek lze považovat jednodušší lineární metody za výhodnější.

3.5 Zhodnocení vybraných vyhledávacích metod

Na základě studia jednotlivých vyhledávacích metod a jejich výsledků testování lze doporučit jejich použití následovně:

- AVL strom (AVL tree): Tato vyhledávací metoda se ukázala být obecně použitelná, a to ve všech případech, ve kterých nelze zjistit nebo neví se bližší specifikace jejího nasazení. Je to metoda, u které se ukázalo, že poskytuje nejrychlejší vyhledávání nad náhodným vstupem klíčů ze všech testovaných vyhledávacích metod. Režie na vkládání a zejména mazání je akceptovatelná za předpokladu, že budeme nad touto metodou provádět především úkony spojené s vyhledáváním. Její nasazení lze doporučit i pro jakýkoli vstup klíčů. Její velikou výhodou je rychlost, nezávislost operací na požadovaných klíčích a nepotřebnost si předem volit velikost struktury. Tuto metodu můžeme bez ostychu nazvat králem vyhledávacích metod alespoň mezi těmito vybranými vyhledávacími metodami.

- Červeno-černý strom (red-black tree): Tato vyhledávací metoda se ukázala být vhodná obzvláště v případech, kdy lze očekávat, že soubor ukládaných položek (každá má svůj klíč) se bude často měnit. Uvažuje-li se náhodný vstup klíčů uzlů, metoda poskytuje nejrychlejší operace pro změny počtů uzlů, což je důsledek méně striktních požadavků na vyváženost, než je tomu u AVL stromu. Režie pro vyhledávání není vysoká. Ve srovnání s AVL stromem její použití záleží jen na prioritách jednotlivých operací. Pokud vstup klíčů bude inklinovat k seřazené posloupnosti, začne se projevovat mírné zpomalení všech operací právě díky méně striktnímu vyvažování stromu. Metoda si zachovává nepotřebnost předem vědět počty uzlů, které budeme chtít vložit. Pokud užijeme přirovnání, že AVL strom je králem vyhledávacích metod, tak červeno-černý strom si zaslouží označení jako nekorunovaný král vyhledávacích metod mezi těmito vybranými vyhledávacími metodami.

- Rozvinutý strom (splay tree): Na této metodě se výrazně projevuje efektivita splay operace, která je vždy vnitřně volána pro vykonání kterékoli základní operace. Zatímco *splay* operace implementovaná způsobem *bottom-up* (princip průchodu zdola nahoru) je na základě testování v praxi málo použitelná a lze ji pouze doporučit nejvýše do školního prostředí k rozšíření obzorů existujících metod, operace *splay top-down* (princip průchodu shora dolů) již má své opodstatnění při znalosti, že vyhledávané klíče nemají náhodné rovnoměrné rozložení, ale jedná se o ne příliš rozsáhlou sadu klíčů, se kterou se bude nejčastěji pracovat. Takovéto často požadované položky budou nalezeny dokonce dřívěji, než je tomu u AVL stromu. Při nesplnění této podmínky se však ukazuje pomalejší. To je v souladu s teorií, která u této vyhledávací metody nezaručuje logaritmickou časovou složitost. Na základě praktického srovnávání různých výsledků si dovolím odhadnout, že tato hranice ne příliš rozsáhlé sady klíčů je někde kolem 15 až 25 % z celkového počtu klíčů. Protože však se mi nepodařilo objektivně navrhnout vstup testovacích klíčů na ne příliš rozsáhlou sadu klíčů, jedná se pouze o spekulativní výsledek. Vzhledem k tomu, že rozvinutý strom má za základ stromovou strukturu, uchovává si vlastnost, že není nutno předem znát počty uzlů, které budeme chtít do ní vložit.

- Přeskakující seznam (skip list): Tato metoda je zajímavá především tím, že se svým chováním řadí do logaritmických vyhledávacích metod, i když jejím základem je prostý lineární jednosměrně vázaný seznam. Při srovnávání jednotlivých metod vyhledávání si stále uchovává pozici nejpomalejší metody, i když není mnohokrát pomalejší než AVL. Její výhodou je snadná pochopitelnost principu chování a jednoduchost průchodu (není to strom, je to seznam). Díky tomu, že úroveň prvku je odvozena na základě pravděpodobnostního rozložení, je předurčena pro srovnatelný počet položek, jako se dává do stromů. I proto lze o ní hovořit jako o optimalizovaném seznamu, protože do lineárního seznamu se vysoké počty prvků z hlavně časových důvodů nevkládají. Na rozdíl od stromových struktur u přeskakujícího seznamu z principu nezáleží na uspořádání klíčů (vše je odvozeno z pravděpodobnostně vygenerované úrovně klíče). Při jejím praktickém použití však kromě ne příliš velké hbitosti se může narazit na požadavek znát asi přibližný počet vkládaných prvků už při inicializaci její struktury (bývá vhodné ji dimenzovat raději na něco více než méně prvků, častěji se totiž přidává).

Kapitola 4

Demonstrační program

4.1 Cíl aplikace

Cílem této aplikace je graficky znázornit jednotlivé vyhledávací metody. Očekává se, že program bude nasazen především pro výukové účely jako doplněk k porozumění vybraných vyhledávacích metod, nikoli jako naučný zdroj, který by v plné šíři měl osvětlovat základní principy např. volených rotací. Předpokládá se tedy předchozí výklad demonstrujícího. Demonstrace bude záměrně tvořena tak, aby každá vyhledávací metoda byla na sobě zcela nezávislá. To umožní při zadání stejných vstupních hodnot do každé datové struktury jejich vzájemné srovnávání, což považuji za velmi přínosné.

4.2 Možnosti uzpůsobení

Protože lze vyjít z předpokladu, že o výuce bude program demonstrován před různě jazykově rozumějící skupinou posluchačů a promítán na plátně v nestejných zobrazovacích podmínkách, je demonstrujícímu k dispozici zvolit si jak jazykové rozhraní programu (automaticky je volen tentýž jazyk jako operační systém), tak i vizuální nastavení, pokud by to původní z jakýchkoliv důvodů nevyhovovalo. Po poměrně dlouhém uvažování, zda změny vizuálních vlastností provádět jen lokálně nad právě zvolenou vyhledávací metodou či nad všemi dostupnými jako je změna jazykové lokalizace, dospěl jsem k závěru, že bude vhodnější změnu provést jen nad vyhledávací metodou, s kterou se aktuálně pracuje (je vybrána), protože pak lze jednotlivé metody od sebe svými různými nastaveními již na první pohled rozlišit i bez znalosti jejich uspořádání a chování. Za tyto vizuální nastavení se považuje promítaná velikost uzlů/prvků, použité písmo a zobrazované barevné rozložení, ke kterým se došlo na základě reálných požadavků běžně se vyskytujících se situací. Jako příklad lze uvést požadavek na změnu barvy uzlu, protože může být defaultně nastavená barva v aktuálních zobrazovacích podmínkách špatně viditelná. Nebo jiná situace, vložili jsme do struktury vysoký počet uzlů, ale rádi bychom ji celou zobrazili do aktuálního okna tak, abychom nemuseli používat posuvníků. To si ale zákonitě žádá i změnu velikosti textu uzlu. Tu je vhodné upozornit na záměrně jednosměrné propojení zobrazovaného textu uzlu a velikosti samotného uzlu. Zatímco změna velikosti uzlu vyvolá úmyslně naprogramovanou změnu velikosti písma na základě přímé úměrnosti, přímá změna textu velikost uzlu nemění. U červeno-černého stromu je (jak lze očekávat) k dispozici volba barvy jak „černého“ tak i „červeného“ uzlu nezávisle.

K možnostem uzpůsobování aplikace zajisté patří i volba, co za text v uzlu zobrazit. Zde

jsou poměrně jednoduché možnosti. Vyšel jsem z toho, že data, která každý uzel s sebou přirozeně nese, nejsou pro ukázku směrodatná. Je tedy zbytečné je při demonstraci zadávat a jakkoli se tu jimi zabývat. Mnoho klíčových vlastností jako je barva uzlu (red-black strom) či vygenerovaná pravděpodobnostní úroveň (skip list) je přímo vidět ze zobrazení. Splay strom je jednoduchý binární strom, takže žádnou speciální vlastnost k dodatečnému zobrazení nemá. Jedině u AVL stromu se vyskytuje vyvažovací faktor, který ne vždy může být na první pohled zřejmý. Proto u této struktury je navíc přidána možnost si jej zobrazit (defaultně není zobrazen). Někdy však uživatele může zajímat pouze struktura uzlů/prvků bez zobrazení klíčů (především při jejich vysokém počtu s malým nastavením velikosti). Je tedy umožněno vypnout zobrazení klíčů, přičemž je-li zatrženo zobrazení vyvažovacího faktoru (případ AVL stromu), jsou tyto hodnoty posunuty doprostřed uzlu na místo klíče pro lepší čitelnost. Dodejme, že každá úprava co zobrazovat/nezobrazovat je na rozdíl od předchozích vlastností globálním požadavkem pro všechny struktury.

4.3 Výběr toolkitu

Volbě vhodného toolkitu[1] pro tvorbu aplikace předcházelo následné uvážení. Vlastnoručně vytvořená knihovna vybraných algoritmů je kompletně napsaná v jazyce C. Pokud bych v rámci kontinuity i pro demonstrační program se měl přidržet nízkoúrovňového jazyka jako je jazyk C, pravděpodobně bych musel sáhnout po OpenGL. Tato knihovna je základní, dostupná nad všemi platformami. Výhodou a současně nevýhodou tohoto řešení by sice byla naprostá kontrola nad dějištěm celého okna, ale velice vysoká náročnost na programování (nutnost se o všechno postarat sám), protože se pracuje jen s grafickými primitivy (bod, úsečka, polygon, ...). Z jiného konce lze uvést např. mnohé aplety (jazyk Java), které při stejném programátorském úsilí jsou schopné poskytnout daleko komfortnější práci díky objektovému konceptu grafických prvků (tlačítko, posuvník, dialog, ...).

Shrňme si požadavky na toolkit.

- Máme vytvořenou knihovnu algoritmů v jazyce C, které by bylo vhodné využít.
- Chtěli bychom pracovat s grafickými objekty.
- Protože program má sloužit pro demonstraci na předem neznámé platformě, toolkit by měl být multiplatformní.
- Volná dostupnost a dosažitelná literatura je podmínkou.

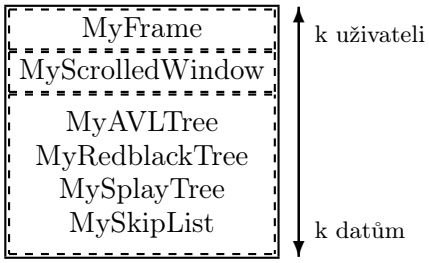
Tyto požadavky z multiplatformních toolkitů nejlépe splnil wxWidgets (jazyk C++), a proto byl v něm program utvořen. Také se v tomto toolkitu poměrně snadno vytváří i multilinguální programy, pokud se s tímto požadavkem již od začátku tvorby programu počítá. Navíc existuje řada materiálů, ze kterých lze při vývoji čerpat. Osobně jsem však zjistil, že největším přínosem k vývoji je kniha [15], která na rozdíl od útržkovitých a částečných textů dostupných na Internetu podává přesně, jasně a srozumitelně způsob používání wxWidgets.

4.4 Logická struktura programu

Program již od svého vzniku vycházel z třívrstvé architektury, která je známa především z databází. Navíc při tvorbě byl program alespoň částečně inspirován z návrhového vzoru

[1] Toolkit je sada programů a připravených nástrojů, které pomáhají při tvorbě programu.

nazývaného Adaptér. Díky tomuto principu je poměrně dobře oddělen vzhled, aplikační logika a data se svými funkcemi k manipulaci. Jak se v těchto principech nabízí, každá vrstva má tak svoji třídní analogii, které představuje schéma 4.1.



Obrázek 4.1: Logická struktura programu

Jak je uvedeno na schématu, nejblíže uživateli je třída nazvaná `MyFrame` (ve stejnojmenném souboru), která je zodpovědná za všechny jednoduché grafické prvky okna. Na této vrstvě je řízena dostupnost grafických prvků a provedeno ošetření korektního vstupu dat od uživatele. Třída `MyScrolledWindow` tvoří jednotné rozhraní pro různé vyhledávací metody, provádí transformaci požadavků vyšší vrstvy k příslušné vyhledávací metodě. Je také zodpovědná za správné umístění a zobrazení výstupu z vyhledávacích metod, korektní nastavení posuvníků a podání hlášení do informačního pole. Na nejnižší vrstvě se nacházejí třídy jednotlivých vyhledávacích metod (`MyAVLTree`, `MyRedblackTree`, `MySplayTree`, `MySkipList`) se svými základními operacemi nad svými daty. V podstatě se jedná o prosté zabalení dříve vytvořené knihovny v jazyce C s přidáním operace pro ohodnocení se za účelem vykreslení. Pro úplnost představovaných tříd uveďme, že třída `MyApp` je hlavní spouštěcí místo celé aplikace (tzv. `main`).

4.5 Algoritmus vykreslení

Při analýze problematiky vykreslování jednotlivých struktur se zjistilo, že seznamy, které jsou založené na lineární struktuře, se daleko snadněji vykreslí než kterékoli stromy, protože stačí jít jednoduše od nejlevějšího prvku a touto jednoduchou lineární strukturou od začátku do konce projít se současným vykreslováním prvků o konstantním kroku ve směru osy x a stejné hodnotě ve směru osy y. To platí i pro skip list, i když tvoří úrovňovou hierarchii dopředných ukazatelů. Platí totiž, že na nejnižší úrovni jsou jednotlivé prvky mezi sebou naprosto stejně svázány jako u každého seznamu. U stromů je však potíž ta, že takový průchod není jednoduchý. I kdybychom dostali posloupnost uzlů, jak jdou zleva doprava za sebou (směr osy x), nebyli bychom schopni jednotlivé uzly správně výškově vykreslit (směr osy y). Proto bylo třeba zvolit jinou taktiku. Ta spočívá v principu čtvercové sítě. Navrhněme tedy algoritmus, jaká je nejmenší možná čtvercová síť, do které můžeme daný strom vykreslit.

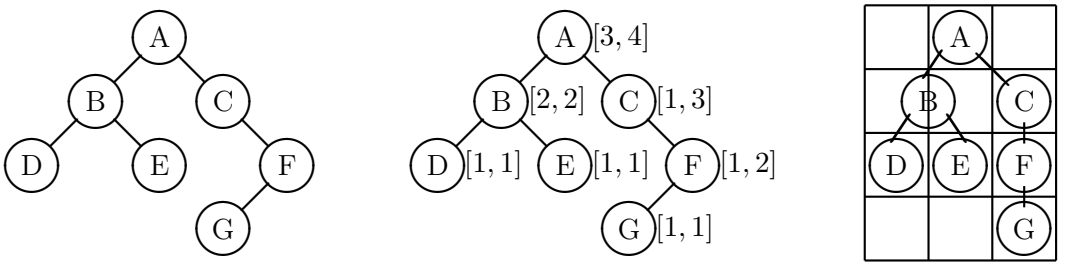
4.5.1 Algoritmus I

Ohodnoťme každý uzel rozměry $[x, y]$. Ty představují minimální rozměry čtvercové sítě pro vykreslení stromu s kořenem právě ohodnocovaného uzlu. Použijme navržená tato pravidla:

- NULL uzel je ohodnocen vždy $[0, 0]$ (neboli uzel nevyžaduje žádné místo)
- Pokud uzel nemá žádného potomka, ohodnoť tento uzel $[1, 1]$ (neboli uzel vyžaduje čtvercové pole 1×1)
- Pokud uzel má nějakého potomka, ohodnoť tento uzel $[\sum_{i=1}^{n} x_i, Max_{i=1}^{n} y_i + 1]$ (neboli uzel vyžaduje čtvercové pole o součtu x-ových velikostí všech svých bezprostředních potomků a nejvyšší y-ovou hodnotu svého bezprostředního potomka zvýšenou o lokální kořen stromu)

Poté, co máme strukturu takto ohodnocenu, nastává fáze přepočítání ohodnocení na pixelové souřadnice obrazovky (s žádaným vykreslením uzlů či uložením pixelových souřadnic pro další zpracování). Na této transformaci je zajímavé jen to, že zatímco na hodnotě transformované y-ové souřadnice se účastní jen počet uzlů na cestě z kořene k vykreslovanému uzlu (nikoli y-ová složka ohodnocení), v x-ové souřadnici uzel umisťujeme doprostřed žádané šířky vykreslovaného uzlu (potřeba x-ová složka ohodnocení) posunuté doprava o součet x-ových rozměrů (šířek) všech bezprostředních levých potomků každého rodiče ležícího v cestě aktuálně vykreslovaného uzlu. V praxi pak máme nějakou proměnnou y, kterou při každém zanoření hlouběji do stromu od kořene zvyšujeme (a naopak snižujeme při vynořování) o zvolenou velikost jednotky čtvercové sítě, a proměnnou x, v které uchováváme průběžný součet šířek všech bezprostředních levých potomků každého rodiče vykreslovaného uzlu. Počty uzlů přímo nepočítáme.

Zatímco pro ohodnocení je nezbytný průchod postorder (požadavek znát ohodnocení všech svých potomků), pro vykreslování lze užít libovolného průchodu, i když preorder by možná byl ze všech nejpřirozenější.



(a) Vzorový strom (b) Ohodnocený strom (c) Vykreslený strom

Obrázek 4.2: Vykreslení stromu podle algoritmu I

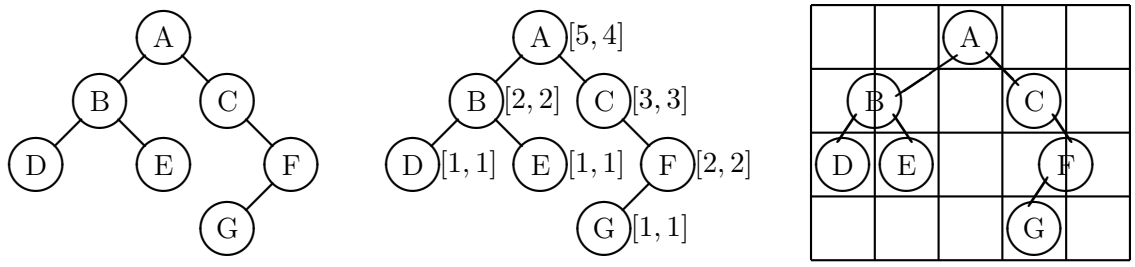
Zkusme tedy strom 4.2(a) ohodnotit podle navržených pravidel „algoritmu I“ (4.2(b)) a zobrazit jej do pomocné čtvercové sítě (4.2(c)). Jak je názorně z obrázku 4.2 vidět, tento princip algoritmu by byl žádaný např.: pro n-ární stromy. Ovšem pro binární stromy nevyhovuje. Navrhněme tedy algoritmus vhodnější, lépe specializovaný pro binární stromy.

4.5.2 Algoritmus II

Ohodnoťme každý uzel rozměry $[x, y]$ dle následujících pravidel:

- NULL uzel je ohodnocen vždy $[0, 0]$ (neboli uzel nevyžaduje žádné místo)
- Pokud uzel má nejvýše jednoho (bezprostředního) potomka, ohodnoť tento uzel $[x_L+x_P+1, Max(y_L, y_P)+1]$ (neboli uzel vyžaduje čtvercové pole o součtu x-ových velikostí všech svých bezprostředních potomků dodatečně rozšířené o šířku „chybějícího“ potomka a nejvyšší y-ovou hodnotu svého libovolného potomka zvýšenou o lokální kořen stromu)
- Pokud uzel má právě oba (bezprostřední) potomky, ohodnoť tento uzel $[x_L + x_P, Max(y_L, y_P) + 1]$ (neboli uzel vyžaduje čtvercové pole o součtu x-ových velikostí všech svých bezprostředních potomků a nejvyšší y-ovou hodnotu svého libovolného potomka zvýšenou o lokální kořen stromu)

Po ohodnocení uzlů můžeme opět zkusit vykreslit. Způsob je obdobný jako v předchozím návrhu. y-ovou souřadnici neustále zvyšujeme o jednotku sítě při každém zanoření hlouběji do stromu od globálního kořene struktury a v x-ové souřadnici kreslíme doprostřed žádané šířky posunuté o součet x-ových rozměrů všech bezprostředních levých potomků každého rodiče ležícího v cestě k aktuálně vykreslovanému uzlu. Nesmíme však opomnět, že v algoritmu ohodnocení jsme při žádném či jediném bezprostředním potomkovi uměle zvyšovali velikost x-ového rozměru o 1 uzel. Proto při získávání rozměru šířky v ose x levého potomka předchůdce, který je právě NULL, provedeme připočítání o jednotku čtvercové sítě, protože jinak, jak plyne z námi vytvořených pravidel ohodnocení, takový uzel vrací rozměr 0.



(a) Vzorový strom (b) Ohodnocený strom (c) Vykreslený strom

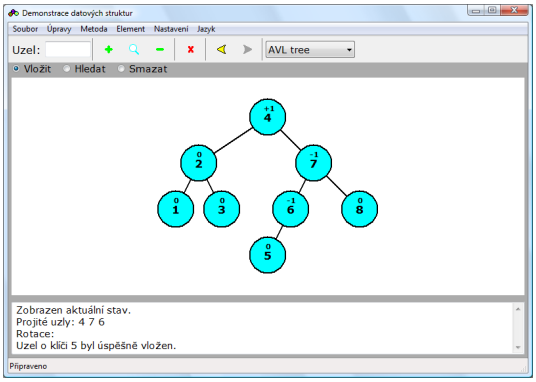
Obrázek 4.3: Vykreslení stromu podle algoritmu II

Zkusme si tedy strom 4.3(a) ohodnotit podle navržených pravidel „algoritmu II“ (4.3(b)) a zobrazit jej do pomocné čtvercové sítě (4.3(c)). Jak je názorně z obrázku 4.3 vidět, tento algoritmus je přesně žádaný pro zobrazování binárního stromu.

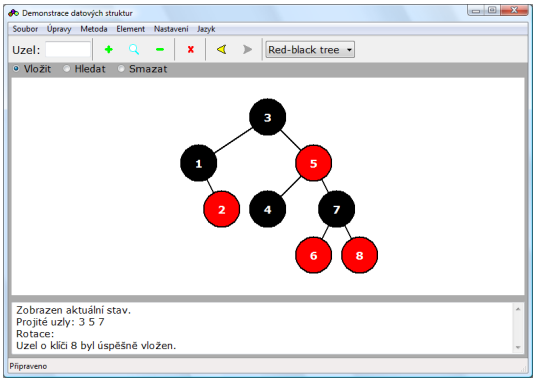
Vzorové kódy obou popsaných algoritmů dále uvádím v příloze.

4.6 Zhodnocení demonstračního programu

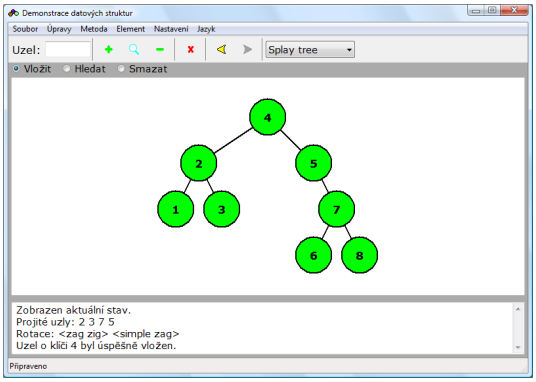
Myslím si, že demonstrační program, jehož několik screenshotů můžeme vidět na obrázku 4.4, napomůže k porozumění a pochopení základních principů zadaných vyhledávacích metod. Ovládací prvky programu jsou intuitivní, při neporozumění jejich významu z názvu prvku je jejich význam vždy blíže popsán ve stavovém řádku. Návrh byl proveden tak, že není velkou potíží dodat kteroukoli další vyhledávací metodu či jazykovou lokalizaci. Proto jsem pro vyzkoušení tohoto rozhraní zkusil vložit i demonstraci AVL stromu, kterou zadání práce přímo nepožaduje, a přidal několik jazykových lokalizací než jen angličtinu a češtinu. Skutečností však zůstává, že na demonstračním programu lze dále pokračovat ve vývoji. Jednou z hlavní oblasti, která by potřebovala zdokonalit, zůstává vizualizace. Bylo by pěkné, kdyby veškeré prováděné proměny stromu byly zobrazovány např. v podobě průběžné animace. Tento neblahý efekt jsem se snažil alespoň částečně zmírnit možností zobrazit si předchozí stav struktury. Na druhou stranu tím, že člověk nevidí přímo vizuálně prováděné změny a má je jen vypsány v textové podobě, se rozvíjí představivost a podněcuje se k vlastnoručnímu rozkreslení předváděné operace a k zamyšlení se nad ní. Jak již jsem uvedl v cílových vlastnostech aplikace, program je multiplatformní a multilinguální pomůckou k pochopení vyhledávacích metod, nikoli kompletním výukovým programem.



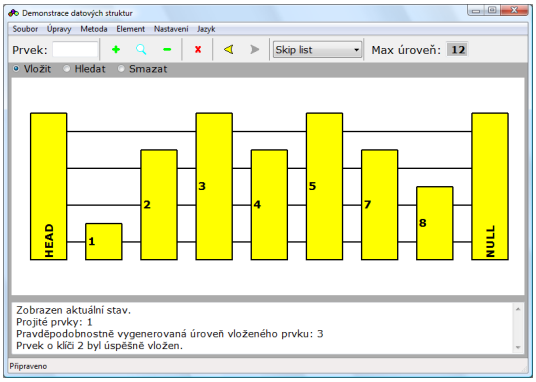
(a) AVL strom



(b) Červeno-černý strom



(c) Rozvinutý strom



(d) Přeskakující seznam

Obrázek 4.4: Vzhled demonstračního programu

Kapitola 5

Závěrečné zhodnocení

Cílem této bakalářské práce bylo nastudovat a provést implementaci zadaných vyhledávacích metod, kterými byly AVL strom (AVL tree), červeno-černý strom (red-black tree), rozvinutý strom (splay tree) a přeskakující seznam (skip list). Následně se měly vytvářené struktury otestovat vlastním programem a porovnat na efektivitu, přičemž správný návrh testování byl také předmětem práce. Na závěr zadání práce se požadovalo vytvořit program sloužící k demonstraci struktur zadaných vyhledávacích metod.

Implementace zadaných metod byla provedena v jazyce C, protože je to programovací jazyk, který je široce rozšířen. Navíc je již od začátku studia zaveden na této fakultě k výuce. Proto ho lze považovat za základní jazyk, který je každému podstatně více srozumitelný, nežli kterýkoli jiný, např. objektový jazyk. Protože jsem shledal za nedostatečné se omezit jen na jediný účel implementace zadaných vyhledávacích metod, jimiž je pouhé zjišťování jejich efektivity, vytvořil jsem z těchto vyhledávacích metod malou knihovnu tak, že je zcela nezávislá na vkládaných datech a porovnávacím klíči a je tedy znovu použitelná v jiných aplikacích, i když je napsána jen v jednoduchém jazyce C.

Při návrhu metody hodnocení jsem dospěl k názoru, že jediným významným atributem pro porovnávání efektivity jednotlivých struktur je měření jejich časového trvání. Vzhledem k obtížím, které se musely při testování jednotlivých operací brát v úvahu, jsou všechny výsledky charakterizující časovou náročnost vztaženy na počty tiků časovače, které operace pro své provedení vyžaduje.

Testováním jednotlivých struktur vyhledávacích metod se potvrdily předpoklady, které vyplývaly z rozboru jednotlivých metod. Dospěl jsem k závěru, že AVL strom je obecně použitelná metoda pro většinu činností, které si kladou za svůj cíl obdržet co nejrychleji výsledek vyhledávání. Společně s přeskakujícím seznamem nezávisí na uspořádání dat. Pokud z jistých důvodů bude počet změn nad strukturou převyšovat počet hledání, lze považovat červeno-černý strom za výhodnější. Bohužel u této vyhledávací metody se již začíná mírně projevovat zpomalení při nerovnoměrném rozložení vkládaných, hledaných či mazaných porovnávacích klíčů. Pokud budeme vědět, že jistá množina dat, která nebude příliš rozsáhlá, bude častěji vyhledávána jak ostatní hodnoty, lze uvažovat o rozvinutém stromu, který se ukazuje při této podmínce být rychlejší i než samotný AVL strom. Pokud však by tato podmínka nebyla zaručena, rozvinutý strom nelze doporučit. Taktéž není vhodné doporučovat přeskakující seznam, protože se v každém měření ve srovnání s ostatními testovanými strukturami jevil jako pomalý. Navíc požaduje znát již od prvního vkládání položek jejich přibližný počet, který výsledně bude v této struktuře uchováván. To však bývá v praxi podstatným handicapem. Blíže k hodnocení jednotlivých vyhledávacích metod se lze dočíst v části 3.5, strana 57.

Je ale skutečností, že principů a jevů, které výsledky měření můžou i výrazně ovlivnit, je velké množství. Některé z nich nemusí jít ani odstranit. Pokud bychom měli být opravdu přesní, měli bychom k měřeným hodnotám zahrnovat a brát v úvahu i další vlivy, které jsem mohl vědomě či nevědomě zanedbat. Protože všechny závěry byly provedeny nad získanou sadou nejistých a neověřených výsledků, bylo by vhodné provést nové zcela nezávislé studium této problematiky, které by ověřilo a potvrdilo či vyvrátilo získané poznatky.

Demonstrační aplikace je napsána v jazyce C++, protože byl využit objektový přístup k její realizaci. Je zaměřena pro demonstraci zadaných vyhledávacích metod a k jejich bližšímu pochopení či dokreslení. Očekává se její nasazení jako doplněk k výuce. Proto má pro zobrazování struktury těchto metod poměrně široké možnosti. Za zmínku také stojí, že je to aplikace multilinguální, která ve svém základním nastavení nabízí možnost volby z anglického, českého a slovenského jazyka. Principiálně však není problém přidat kterýkoli myslitelný jazyk taktéž jako kteroukoli vyhledávací metodu splňující jisté rozhraní. Protože se předem neví, na jaké platformě bude aplikace běžet, byl zvolen platformově nezávislý toolkit wxWidgets. Aplikace je tedy multiplatformní. Jiným směrem řešení by sice byla volba webové aplikace (např. applet), ale protože nelze počítat s připojením k Internetu při probíhající demonstraci, byl tento způsob zavrhnut. Ve vývoji na demonstračním programu však lze úspěšně pokračovat dál. Kromě zvyšování a rozšiřování funkcionality programu by bylo vhodné se více zaměřit např. na oblast vykreslování. Možnost ukládat a načítat jistou konfiguraci struktury dané metody by určitě uživatele také potěšila.

První seznámení s tímto zadáním proběhlo na konci 4. semestru, kdy mi byla nabídnuta různá zajímavá témata ke zpracování coby bakalářská práce. Na realizaci této práce se začalo hned v 5. semestru, kdy byly lépe upřesněny její požadavky a cíle. Následně bylo obdrženo její zadání. Na programovém zpracování se pracovalo již od 5. semestru, do konce března 6. semestru byla veškerá programová část hotova. Od prosince s vytvářením programové části postupně začínala souběžná tvorba této technické zprávy. Její zpracování trvalo až do samotného odevzdání celé bakalářské práce.

Literatura

[1] ADELSON-VELSKII, G. M.; LANDIS, E. M.: An Algorithm for the Organization of Information. *Soviet Mathematics – Doklady*, ročník 3, 1962: s. 1259–1263.

[2] BAYER, R.: *Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms*. Acta Informatica, první vydání, 1972, s. 290–306.

[3] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.: *Introduction to Algorithms*. Massachusetts: MIT Press, druhé vydání, 2001, ISBN 0-262-03293-7, 1180 s.

[4] Dr. Dobb's Journal: *Algorithms and Data Structures: Dr. Dobb's Essential Books*. Druhé vydání, 1999, [CD-ROM], Zdroj obsahuje 10 nejvýznamnějších monografií.

[5] DVORSKÝ, J.: *Algoritmy I*. Studijní materiál, Katedra informatiky FEI VŠB TU Ostrava, Únor 2007, [online], [cit. 2008-04-20].
URL http://www.cs.vsb.cz/dvorsky/Download/Algoritmy/ZakladyAlgoritmizace28022007.pdf

[6] GUIBAS, L. J.; SEDGEWICK, R.: *A Dichromatic Framework for Balanced Trees*. Proceedings of the 19th Annual Symposium on Foundations of Computer Science. IEEE Computer Society, 1978, s. 8–21.

[7] GURARI, E.: *Introduction to Algorithms*. The Ohio State University – Department of Computer Science and Engineering, Srpen 1999, [online], [cit. 2008-04-02].
URL http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch10.html

[8] HONZÍK, J. M.: *Algoritmy – studijní opora*. Studijní materiál, FIT VUT v Brně, Říjen 2007, 4. verze, [online], [cit. 2008-04-22].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IAL-IT/texts/Opora-IAL-2007-05-RP-verze-4.pdf

[9] KRESLÍKOVÁ, J.: *Základy programování – úvod do algoritmizace*. Studijní materiál, FIT VUT v Brně, Říjen 2005, [online], [cit. 2008-04-02].
URL https://www.fit.vutbr.cz/study/courses/IZP/private/1c_alg-uvod.pdf

[10] MINTĚL, T.: *Vyhledávání v AVL stromech v jazyce C*. Bakalářská práce, FIT VUT v Brně, Brno, 2007.

[11] *PC Timers*. [online], [cit. 2008-03-20].
URL http://www.mindcontrol.org/~hplus/pc-timers.html

[12] PUGH, W.: *Skip Lists: A Probabilistic Alternative to Balanced Trees*. Březen 1999, [online], úryvek z původní studie, [cit. 2008-03-24].
URL ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

[13] *Simple PC Timer using QueryPerformanceCounter*. [online], [cit. 2008-03-20].
URL http://www.mindcontrol.org/~hplus/misc/simple-timer.html

[14] SLEATOR, D. D.; TARJAN, R. E.: Self-Adjusting Binary Search Trees. *Journal of the Association for Computing Machinery*, ročník 32, č. 3, 1985: s. 652–686, [online], [cit. 2008-04-02].
URL http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf

[15] SMART, J.; HOCK, K.; CSOMOR, S.: *Cross-Platform GUI Programming with wxWidgets*. Pearson Education, Inc., 2006, ISBN 0-13-147381-6, 700 s.

[16] TÖPFLER, P.: *Algoritmy a programovací techniky*. Prometheus, první vydání, 1995, ISBN 80-85849-83-6, 300 s.

[17] WALKER, J.: *Skip Lists*. Eternally Confuzzled Team, [online], [cit. 2008-04-02].
URL
http://eternallyconfuzzled.com/tuts/datastructures/jsw_tut_skip.aspx

[18] Wikipedia: *AVL tree*. [online], poslední aktualizace 13. 04. 2008, [cit. 2008-04-22].
URL http://en.wikipedia.org/wiki/AVL_tree

[19] Wikipedia: *Context switch*. [online], poslední aktualizace 23. 03. 2008, [cit. 2008-04-04].
URL http://en.wikipedia.org/wiki/Context_switch

[20] Wikipedia: *IEEE 754*. [online], poslední aktualizace 25. 03. 2008, [cit. 2008-04-06].
URL http://en.wikipedia.org/wiki/IEEE_754

[21] Wikipedia: *Long double*. [online], poslední aktualizace 18. 04. 2007, [cit. 2008-04-26].
URL http://en.wikipedia.org/wiki/Long_double

[22] Wikipedia: *Pentium III*. [online], poslední aktualizace 27. 03. 2008, [cit. 2008-04-04].
URL http://cs.wikipedia.org/wiki/Pentium_III

[23] Wikipedia: *RDTSC*. [online], poslední aktualizace 21. 03. 2008, [cit. 2008-03-24].
URL http://en.wikipedia.org/wiki/RDTSC

[24] Wikipedia: *Red-black tree*. [online], poslední aktualizace 12. 04. 2008, [cit. 2008-04-22].
URL http://en.wikipedia.org/wiki/Red-black_tree

[25] Wikipedia: *Skip list*. [online], poslední aktualizace 28. 04. 2008, [cit. 2008-04-30].
URL http://en.wikipedia.org/wiki/Skip_list

[26] Wikipedia: *Splay tree*. [online], poslední aktualizace 21. 04. 2008, [cit. 2008-04-29].
URL http://en.wikipedia.org/wiki/Splay_tree

Dodatek A

Příloha 1 – Ukázka použití vytvořené knihovny vyhledávacích metod

```
#include <stdio.h>
#include <stdlib.h>
// přilinkování hlaviček vyhledávacích metod (pokud s nimi pracujeme)
#include "AVLTree.h"
#include "RedBlackTree.h"
#include "SplayTree.h"
#include "SkipList.h"
```

Definování struktury dat ukládaných do každého uzlu (pro metody, s kterými hodláme pracovat):

```
// datová struktura uzlu AVL stromu (AVL tree)
struct AVLDataItem {
  int myData;  // zde si uživatel vkládá jakékoli položky,
  char myData2[15];  // libovolného počtu a datového typu
      /* ... */
};
// datová struktura uzlu červeno-černého stromu (red-black tree)
struct RBTDataItem {
      /* ... */
};
// datová struktura uzlu rozvinutého stromu (splay tree)
struct SPTDataItem {
      /* ... */
};
// datová struktura prvku přeskakujícího stromu (skip list)
struct SKLDataItem {
      /* ... */
};
```

Vytvoření porovnávací funkce nad typem klíče (zvoleným uživatelem knihovny), její deklarace je pevně dána:

```
int compare(const void *i, const void *j)
```

```
{
  int a = *(int *)i;  // podle typu klíče provádíme příslušnou dereferenci
  int b = *(int *)j;
  // návratová hodnota dle výsledku porovnání a, b;
  // návratové hodnoty: menší 0, 0, větší 0 (nevyžaduje se striktně -1, 0, +1)
  return (a==b)? 0 : ((a<b)? -1 : +1);
}
```

Pokud požadujeme někdy vypsat danou strukturu, vytváříme funkci tisku položek uzlu (obdrženého přes ukazatel) do předaného proudu (její deklarace je pevně dána):

```
void AVLPrintNodeDataItem(FILE *stream, const tAVLNode *node)
{
  // výtisk klíče (dereference podle jeho typu), dat definovaných
  // ve struktuře uzlu (int myData; a char myData2[15]; ... ) a váhy uzlu
  fprintf(stream, "[%d, %d, %s, %d]", *(int*)node->key,
    node->data->myData, node->data->myData2, node->weight);
}
void RBTPrintNodeDataItem(FILE *stream, const tRBTNode *node)
{
  // výtisk klíče (dereference podle jeho typu), dat definovaných
  // ve struktuře uzlu (...) a barvy uzlu
  fprintf(stream, "[%d, %s]", *(int*)node->key,
    (node->colour == RED)? "RED" : "BLACK");
}
void SPTPrintNodeDataItem(FILE *stream, const tSPTNode *node)
{
  // výtisk klíče (dereference podle jeho typu) a dat definovaných
  // ve struktuře uzlu (...)
  fprintf(stream, "[%d]", *(int*)node->key);
}
void SKLPrintNodeDataItem(FILE *stream, const tSKLNode *node)
{
  // výtisk klíče (dereference podle jeho typu), dat definovaných
  // ve struktuře uzlu (...) a pravděpodobnostně vygenerované úrovně uzlu
  fprintf(stream, "[%d, %d]", *(int*)node->key, node->actHeight);
}
```

Vstupní bod programu:

```
int main(void)
{
  int myKey=0;  // právě zde jsme si určili, že klíč uzlu bude typu int
  int ret;  // pomocná proměnná - návratová hodnota
  int maxLevel = 16;  // stanovení maximální úrovně (pro skip list)

  struct AVLDataItem myDataAVLTree = {0, "TEXT IN ARRAY"};  // struktura dat
  struct RBTDataItem myDataRedblackTree = { /* ... */ };
```

```
struct SPTDataItem myDataSplayTree = { /* ... */ };
struct SKLDataItem myDataSkipList = { /* ... */ };

tAVLTree AVLtree;  // AVL tree - vstupní bod (kořen) do této struktury
tRedblackTree RBtree;  // red-black tree - vstupní bod (kořen) do této struktury
tSplayTree SPtree;  // splay tree - vstupní bod (kořen) do této struktury
tSkipList SKlist;  // skip list - vstupní bod (kořen) do této struktury
```

Inicializace příslušné struktury (předat správně velikost klíče, velikost datové struktury a ukazatel na vytvořenou porovnávací funkci):

```
AVLInit(&AVLtree, sizeof(myKey), sizeof(struct AVLDataItem), &compare);
RBTInit(&RBtree, sizeof(myKey), sizeof(struct RBTDataItem), &compare);
SPTInit(&SPtree, sizeof(myKey), sizeof(struct SPTDataItem), &compare);
ret = SKLInit(&SKlist, maxLevel, sizeof(myKey),
        sizeof(struct SKLDataItem), &compare);
if(ret == EXIT_FAILURE){ /* chyba alokace skip list při inicializaci */ }
```

Vložení klíče a dat do příslušné struktury:

```
ret = AVLInsert(&AVLtree, &myKey, &myDataAVLTree);
if(ret == EXIT_FAILURE){ /* chyba alokace AVL tree při vkládání */ }
ret = RBTInsert(&RBtree, &myKey, &myDataRedblackTree);
if(ret == EXIT_FAILURE){ /* chyba alokace red-black tree při vkládání */ }
ret = SPTInsert(&SPtree, &myKey, &myDataSplayTree);
if(ret == EXIT_FAILURE){ /* chyba alokace splay tree při vkládání */ }
ret = SKLInsert(&SKlist, &myKey, &myDataSkipList);
if(ret == EXIT_FAILURE){ /* chyba alokace skip list při vkládání */ }
```

Aktualizace dat v příslušné struktuře:

```
ret = AVLActualize(&AVLtree, &myKey, &myDataAVLTree);
if(ret == EXIT_FAILURE){ /* uzel v AVL tree nenalezen */ }
ret = RBTActualize(&RBtree, &myKey, &myDataRedblackTree);
if(ret == EXIT_FAILURE){ /* uzel v red-black tree nenalezen */ }
ret = SPTSearch(&SPtree, &myKey, &myDataSplayTree);
if(ret == EXIT_FAILURE){ /* uzel ve splay tree nenalezen */ }
ret = SKLActualize(&SKlist, &myKey, &myDataSkipList);
if(ret == EXIT_FAILURE){ /* uzel ve skip list nenalezen */ }
```

Vyhledání uzlu a obdržení dat v příslušné struktuře:

```
ret = AVLSearch(&AVLtree, &myKey, &myDataAVLTree);
if(ret == EXIT_FAILURE){ /* uzel ve AVL tree nenalezen */ }
ret = RBTSearch(&RBtree, &myKey, &myDataRedblackTree);
if(ret == EXIT_FAILURE){ /* uzel ve red-black tree nenalezen */ }
ret = SPTSearch(&SPtree, &myKey, myDataSplayTree);
```

```
  if(ret == EXIT_FAILURE){ /* uzel ve splay tree nenalezen */ }
  ret = SKLSearch(&SKlist, &myKey, &myDataSkipList);
  if(ret == EXIT_FAILURE){ /* uzel ve skip list nenalezen */ }
```

Tisk příslušné struktury na „stdout“ (standardní výstup – obrazovku), první parametr je volba datového proudu, druhý volba datové struktury a třetí parametr je ukazatel na vytvořenou funkci tisku uzlu (obdržen přes ukazatel):

```
  ret = AVLPrint(stdout, &AVLtree, &AVLPrintNodeDataItem);
  if(ret == EXIT_FAILURE){ /* nedostatek paměti k výpisu AVL tree struktury */ }
  ret = RBTPrint(stdout, &RBtree, &RBTPrintNodeDataItem);
  if(ret == EXIT_FAILURE){ /* nedostatek paměti k výpisu red-black tree struktury */ }
  ret = SPTPrint(stdout, SPtree, &SPTPrintNodeDataItem);
  if(ret == EXIT_FAILURE){ /* nedostatek paměti k výpisu splay tree struktury */ }
  SKLPrint(stdout, SKlist, &SKLPrintNodeDataItem);
```

Vymazání uzlu v příslušné struktuře:

```
  ret = AVLDelete(&AVLtree, &myKey);
  if(ret == EXIT_FAILURE){ /* uzel v AVL tree nenalezen */ }
  ret = RBTDelete(&RBtree, &myKey);
  if(ret == EXIT_FAILURE){ /* uzel v red-black tree nenalezen */ }
  ret = SPTDelete(&SPtree, &myKey);
  if(ret == EXIT_FAILURE){ /* uzel ve splay tree nenalezen */ }
  ret = SKLDelete(&SKlist, &myKey);
  if(ret == EXIT_FAILURE){ /* uzel ve skip list nenalezen */ }
```

Uvolnění příslušné struktury:

```
  AVLDispose(&AVLtree);
  RBTDispose(&RBtree);
  SPTDispose(&SPtree);
  SKLDispose(&SKlist);
}  // main()
```

Dodatek B

Příloha 2 – Implementace algoritmu vykreslení stromové struktury

V dalším textu předpokládejme, že máme definovány tyto struktury:

```
typedef struct size{  // struktura uspořádané dvojice [x,y]
  int x;
  int y;
}tSize;

typedef struct node{  // struktura uzlu
  void *key;  // klíč, dle kterého se vyhledává
  struct DataItem *data;  // ukazatel na strukturu dat - tvoří uživatel
  struct node *left;  // ukazatel na levý podstrom
  struct node *right;  // ukazatel na pravý podstrom
  int x, y;  // tento lokální kořenový uzel vyžaduje [x,y] jednotek pro vykreslení
  int width;  // tento uzel má tuto šířku (v pixelech)
} tNode;

unsigned shift = 20;  // jednotková hrana čtvercové sítě (počet pixelů)
```

B.1 Implementace algoritmu I

Ohodnocení I:

```
// provede prostorové ohodnocení binárního stromu ve stylu n-árního stromu
tSize evaluate(tNode *N)
{  // ohodnocení je založeno na rekurzivním průchodu stromem - postorder
  tSize sizeLeft;  // rozměry vrácené zleva
  tSize sizeRight;  // rozměry vrácené od zprava
  tSize sizeReturn;  // rozměry, které vrátit
  if(N == NULL){
    tSize size = {0, 0};  // NULL list uzel nemá žádné rozměry pro vykreslení
    return size;
  }
  sizeLeft = evaluate(N->left);  // zjisti rozměry zleva
```

```
  sizeRight = evaluate(N->right);  // zjisti rozměry zprava
  // vrať součet bezprostředních potomků nebo velikost tohoto uzlu (1),
  // je-li šířka tohoto uzlu větší jak součet šířek potomků
  sizeReturn.x = (sizeLeft.x + sizeRight.x > 1) ? sizeLeft.x + sizeRight.x : 1;
  // vrať větší výšku potomka zvětšené o výšku tohoto uzlu
  sizeReturn.y = ((sizeLeft.y > sizeRight.y) ?
                  sizeLeft.y : sizeRight.y) + 1;
  // poznačení ohodnocení do uzlu
  N->x = sizeReturn.x;
  N->y = sizeReturn.y;
  // tento uzel pro vykreslení svého stromu má sizeReturn prostorové nároky
  return sizeReturn;
} // evaluate()
```

Transformace I:

```
// provede transformaci ohodnocení na pixelové hodnoty
void evaluatePixelTransform(tNode *N, int pixelx, int pixely)
{  // ohodnocení je založeno na rekurzivním průchodu stromem - preorder
  if(N == NULL)  // je-li NULL, není co přepočítat
    return;
  N->width = N->x * shift;  // šířka v pixelech
  /**/ uložení pixelových hodnot pro vykreslení uzlu nebo přímé vykreslení
  /**/ N->x = N->width/2 + pixelx;  // přepočet x souřadnice pro vykreslení na střed
  /**/ N->y = pixely;  // přepočet y souřadnice (výška)
  // transformace na levý podstrom
  evaluatePixelTransform(N->left, pixelx, pixely+shift);
  // posunutí počátku x souřadnice (šířka) o šířku levého sourozence
  if(N->left != NULL)
    pixelx += N->left->width;  // je levý uzel o nějaké šířce (>=1)
  // transformace na pravý podstrom
  evaluatePixelTransform(N->right, pixelx, pixely+shift);
}
```

B.2 Implementace algoritmu II

Ohodnocení II:

```
// provede prostorové ohodnocení stromu
tSize evaluate(tNode *N)
{  // ohodnocení je založeno na rekurzivním průchodu stromem - postorder
  tSize sizeLeft;  // rozměry vrácené zleva
  tSize sizeRight;  // rozměry vrácené zprava
  tSize sizeReturn;  // rozměry, které vrátit
  if(N == NULL){
    tSize size = {0, 0};  // NULL list uzel nemá žádné rozměry pro vykreslení
    return size;
  }
```

```
  sizeLeft = evaluate(N->left);  // zjisti rozměry zleva
  sizeRight = evaluate(N->right);  // zjisti rozměry zprava
  // nemá-li uzel oba potomky, vrátit požadovanou šířku + 1, jinak jen tuto šířku
  sizeReturn.x = (sizeLeft.x==0 || sizeRight.x==0) ?
                  sizeLeft.x + sizeRight.x +1 : sizeLeft.x + sizeRight.x;
  // vrať větší výšku potomka zvětšené o výšku tohoto uzlu
  sizeReturn.y = ((sizeLeft.y > sizeRight.y) ?
                  sizeLeft.y : sizeRight.y) + 1;
  // poznačení ohodnocení do uzlu
  N->x = sizeReturn.x;
  N->y = sizeReturn.y;
  // tento uzel pro vykreslení svého stromu má sizeReturn prostorové nároky
  return sizeReturn;
} // evaluate()
```

Transformace II:

```
// provede transformaci ohodnocení na pixelové hodnoty
void evaluatePixelTransform(tNode *N, int pixelx, int pixely)
{  // ohodnocení je založeno na rekurzivním průchodu stromem - preorder
  if(N == NULL)  // je-li NULL, není co přepočítat
    return;
  N->width = N->x * shift;  // šířka v pixelech
  /**/ uložení pixelových hodnot pro vykreslení uzlu nebo přímé vykreslení
  /**/ N->x = N->width/2 + pixelx;  // přepočet x souřadnice pro vykreslení na střed
  /**/ N->y = pixely;  // přepočet y souřadnice (výška)
  // transformace na levý podstrom
  evaluatePixelTransform(N->left, pixelx, pixely+shift);
  // posunutí počátku x souřadnice (šířka) o šířku levého sourozence
  if(N->left != NULL)
    pixelx += N->left->width;  // je levý uzel o nějaké šířce (>=1)
  else  // levý uzel je NULL (není co vykreslit)
    pixelx += shift;  // posun o shift, aby se uzel vykreslil vpravo
  // transformace na pravý podstrom
  evaluatePixelTransform(N->right, pixelx, pixely+shift);
}
```

Dodatek C

Příloha 3 – Datové CD

Přiložené datové CD obsahuje:

- Zdrojové texty práce v elektronické podobě
- Zdrojové kódy implementace jednotlivých celků
    1. kód implementace zadaných vyhledávacích metod při překladu sestavovaných do knihovny s krátkým programem pro rozhodnutí o efektivní alokaci paměti a jednoduchým programem pro experimentální zjištění maximální rekurzivní hloubky
    2. hlavní testovací program pro měření doby trvání jednotlivých operací v různých metodách (užívá vytvořenou knihovnu)
    3. platformově nezávislý demonstrační program jednotlivých vyhledávacích metod
- Vysázený text práce a přeložené výsledky prací z jednotlivých částí