Univerzita Pardubice
Fakulta elektroniky a informatiky

Využití datových struktur v jazyce Java a JavaFX.

Lukáš Pírko

Bakalářská práce

2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2017/2018

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Lukáš Pírko**
Osobní číslo:	**I15120**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Využití datových struktur v jazyce Java a JavaFX**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

V rámci teoretické části práce se student zaměří na popis základních datových struktur používaných v programovacím jazyce Java a JavaFX. Provede jejich přehled a zaměří se na jejich vnitřní implementaci tak, aby odlišil případy užití těchto datových struktur. Studovanými strukturami budou především ArrayList, LinkedList, Vector, Set, Map (vč. jeho základních implementací) a dále Observable varianty struktur v JaveFX.
V praktické části práce student provede implementaci nejméně pěti datových struktur prezentovaných v teoretické části. Na implementovaných strukturách provede výkonové srovnání při parametrizaci generovanými hodnoty typu Integer a String (tj. dvě varianty testů) a šíří testovaných operací 10 až 100.000, při logaritmickém měřítku a deseti až dvaceti měřených hodnotách. Základními testovanými operacemi budou: zápis, přístup dle indexu, přístup dle klíče, odebrání, plnění.

Rozsah grafických prací:

Rozsah pracovní zprávy: **35 - 45 stran**

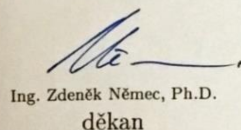
Forma zpracování bakalářské práce: **tištěná**

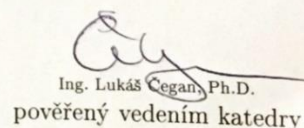
Seznam odborné literatury:

1. LEWIS, H. R., DENENBERG, L. Data structures and their algorithms. Berkley, Adison-Wesley, 1997.
2. KNUTH, D. E.: Umění programování - Základní algoritmy, Brno, Computer Press 2008, ISBN: 978-80-251-2025-5
3. WRÓBLEWSKI, Piotr. Algoritmy : datové struktury a programovací techniky / Piotr Wróblewski. Marek Michalek, Bogdan Kiszka. Vyd 1. Brno : Computer Press, 2004. 351 s. ISBN 80-251-0343-9.
4. KEOGH, Jim; DAVIDSON, Ken. Datové struktury bez předchozích znalostí : průvodce pro samouky. Vyd 1. Brno : Computer Press, 2006. 223 s. ISBN 80-251-0689-6.
5. PROKOP, Jiří. Algoritmy v jazyku C a C++ : praktický průvodce. Vyd 1. Praha : Grada, 2009. 153 s. ISBN 978-80-247-2751-6.

Vedoucí bakalářské práce: **Ing. Josef Brožek**
Katedra informačních technologií

Datum zadání bakalářské práce: **31. října 2017**
Termín odevzdání bakalářské práce: **12. května 2018**


Ing. Zdeněk Němec, Ph.D.
děkan


Ing. Lukáš Čegan, Ph.D.
pověřený vedením katedry

V Pardubicích dne 20. března 2018

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 5. 12. 2018

Lukáš Pírko

Rád bych poděkoval vedoucímu práce Ing. Josefu Brožkovi za vstřícný přístup a cenné rady při zpracovávání bakalářské práce.

ANOTACE

Cílem práce je popsat několik základních datových struktur využívaných v programovacím jazyce Java a JavaFX. Studovanými strukturami budou především ArrayList, LinkedList, Vector, Set, Map.

Praktická část se zaměří na implementaci pěti datových struktur, které budou prezentovány v teoretické části. Na implementovaných strukturách bude provedeno výkonové srovnání. Testovanými operacemi budou: zápis, přístup dle indexu, přístup dle klíče, odebírání, plnění.

KLÍČOVÁ SLOVA

Datové struktury, algoritmus, lineární seznam, lineární struktury, pole, strom, Java, JavaFX, složitost algoritmu, efektivita algoritmu, složitost

TITLE

Using data structures in Java and JavaFX

ANNOTATION

The objective of the thesis is to describe several basic data structures used in the programming language Java and JavaFX. Studied structures will be primarily ArrayList, LinkedList, Vector, Set, Map.

The practical part will focus on the implementation of five data structures, which will be presented in the theoretical part. Performance benchmarking will be used on the implemented structures. Following operation will be tested: write, index access, key access, removal, filling.

KEYWORDS

Data structures, algorithm, list, linear structure, array, tree, Java, JavaFX, algorithm complexity, algorithm efficiency, complexity.

OBSAH

SEZNAM ZKRATEK 9

SEZNAM OBRÁZKŮ 10

SEZNAM TABULEK 11

ÚVOD 12

1 SLOŽITOSTI ALGORITMŮ 13

1.1 Složitost 13

1.2 Asymptotická složitost 14

1.3 Výkonnost 14

1.4 Složitost programu 15

2 DATOVÉ STRUKTURY 16

2.1 ArrayList 16

2.2 Vector 17

2.3 LinkedList 18

2.4 Set 19

2.5 Map 20

2.6 Observable 21

3 POUŽITÉ NÁSTROJE A PROSTŘEDÍ 22

3.1 Použité prostředí 22

3.2 Použitý hardware 22

4 METODIKA SROVNÁNÍ ALGORITMŮ 24

4.1 Výběr struktur 24

4.2 Programovací paradigmata 25

4.3 Testovací sestava 26

4.4 Stanovení předpokladů 28

4.5 Implementace 28

5 VÝSLEDKY ANALÝZY 32

5.1 ArrayList 32

5.2 Vector 37

5.3 LinkedList 37

5.4 Map 40

5.5 Set 42
5.6 Zhodnocení předpokladů 42
ZÁVĚR 44
POUŽITÁ LITERATURA 45

SEZNAM ZKRATEK

RTOS	Real time systém
MSDNAA	Zkratka pro licenční program od Microsoftu
OEM	Způsob licencování softwaru
GNU GPL	Všeobecná veřejná licence GNU
IDE	Integrované vývojové prostředí
ISIC	Zkratka pro mezinárodní studentský průkaz
CPU	Procesor počítače
GPU	Grafický procesor počítače
HW	Hardware
JVM	Java virtual machine
RAM	Operační paměť

SEZNAM OBRÁZKŮ

Obrázek 1: Příklad vyhledávání v poli 13
Obrázek 2: Režie antivirového programu 27
Obrázek 3: Ukázka výstupu konzole 29
Obrázek 4: Ukázka základního členění logiky testů v programu 29
Obrázek 5: Příklad volání metod testů v rozdělených do logických celků 30
Obrázek 6: Náhled na konkrétní implementaci testu pro jednu operaci 31
Obrázek 7: ArrayList vkládání na konec seznamu 32
Obrázek 8: ArrayList rozdíl mezi datovými typy 33
Obrázek 9: ArrayList porovnání s nativním ArrayListem 33
Obrázek 10: ArrayList náhodné vložení prvku do pole 34
Obrázek 11: ArrayList porovnání nativní implementace s vlastní optimalizovanou 34
Obrázek 12: Stav metody před optimalizací 35
Obrázek 13: Časová amortizace vkládání na začátek ArrayListu 35
Obrázek 14: ArrayList vs. DoublyLinkedList metoda přidání na konec 36
Obrázek 15: Porovnání času k přístupu mezi nativním a vlastním polem 36
Obrázek 16: Průběh amortizace při odebírání z konce seznamu 37
Obrázek 17: Porovnání vlastní implementace s nativní implementací Vectoru 37
Obrázek 18: Přehled implementací LinkedListu 38
Obrázek 19: Náhodný přístup k prvku v seznamu oproti ArrayListu 38
Obrázek 20: Vkládání na náhodnou pozici v seznamu 39
Obrázek 21: Odebírání posledního prvku z LinkedListu vs. Nativní LinkedList 39
Obrázek 22: Odebírání prvního prvku z LinkedListu vs. Nativní LinkedList 40
Obrázek 23: Operace přidání TreeMap vs. Nativní TreeMap 40
Obrázek 24: Operace čtení přiblížení začátku průběhu TreeMap vs. Nativní TreeMap 41
Obrázek 25: Průběh operace čtení v plné šířce TreeMap vs. Nativní TreeMap 41
Obrázek 26: Odebírání kořene TreeMap vs Native TreeMap 41
Obrázek 27: Odebírání kořene TreeSet vs Native TreeSet 42
Obrázek 28: Odebírání kořene MyTreeMap vs MyTreeSet 42

SEZNAM TABULEK

Tabulka 1: Tabulka nárůstu složitostí 14
Tabulka 2: Vliv utříděnosti na složitost 25
Tabulka 3: Stav řešení 26

ÚVOD

Cílem této práce je porovnání vybraných datových struktur, při základních operacích jako je vkládání, odebírání, přístup dle indexu a přístup dle klíče. Práce dále bude zaměřena na problematiku složitosti těchto datových struktur, způsob využití v různých systémech, jejich výhody a nevýhody.

Práce je rozdělena do pěti kapitol. První kapitola vymezuje základní problematiku datových struktur a pojednává o způsobu hodnocení daných datových struktur (resp. složitosti daných operací).

Druhá kapitola je zaměřena na popis vybraných datových struktur a částečné přiblížení jejich vnitřní implementace. Dále je v této kapitole zmíněno, jakým způsobem si datové struktury v sobě strukturují svá data a jak jejich vnitřní implementace ovlivňuje složitost daných operací.

Vzhledem k zaměření této práce je nutné, aby v třetí kapitole bylo rozebráno, na jakém hardware budou výpočty prováděny, jaké budou použita vývojová prostředí, nástroje na tvorbu a vizualizaci grafů, díky kterým bude přehlednější a snadnější pochopení chování testovaných struktur.

Ve čtvrté kapitole bude přiblížena metodika srovnání algoritmů, především bude určeno, co bude srovnáno, jakým způsobem budou data porovnávána a v neposlední řadě bude představeno jádro celé práce a tím je vlastní software na porovnávání datových struktur.

Poslední kapitola je zaměřena na výslednou analýzu, porovnání se stanovenými předpoklady a přehled výsledků vizualizovaných grafů společně s komentáři.

1 SLOŽITOSTI ALGORITMŮ

Složitost problému je složitost asymptoticky nejlepšího možného algoritmu, který řeší daný problém. Kapitola vycházela z (Lewis a Denenberg 1991, Wróblewski 2004, Mehta a Sahni 2018)

1.1 Složitost

Jednou z důležitých vlastností algoritmu je jeho časová náročnost, podle výpočtů provedených na základě daného algoritmu. Tato časová náročnost se nezískává měřením doby výpočtu pro různá data, ale analýzou algoritmu. Výsledkem takové analýzy je časová složitost algoritmu.

Časová složitost algoritmu vyjadřuje závislost času potřebného pro provedení výpočtu na velikosti vstupních dat. Čas se neměří v sekundách, ale počtem provedených operací, přičemž trvání každé operace se chápe jako bezrozměrná jednotka. Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale také na konkrétních hodnotách. Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném (středním) případě.

Na obrázku 1 lze vidět příklad sekvenčního procházení a vyhledávání v poli.

```
for (int i =0 ; i < array.length; i++) {
    if (var == array[i]) return true;
}
return false;
```

Obrázek 1: Příklad vyhledávání v poli

Nejlepší případ – první prvek má hledanou hodnotu

Nejhorší případ – žádný prvek nemá hledanou hodnotu

Ovšem přesné určení počtu operací při analýze složitosti algoritmu bývá velmi složité, především určení počtu operací v průměrném případě, bývá skoro nemožné. Proto se uchylujeme pouze na analýzu nejhoršího případu.

Většinou nás nezajímá konkrétní počty operací pro různé rozsahy vstupních dat n, ale tendence jejich růstu při zvyšujícím se n.

Složitost algoritmu udává, jak je daný algoritmus rychlý (kolik je schopen provést operací) vzhledem k vstupním datům. Ke klasifikaci algoritmů se obvykle používá

asymptotická složitost, což je rozdělení algoritmů do tříd složitostí, u kterých platí, že od určité velikosti dat, je algoritmus dané třídy vždy pomalejší než algoritmus třídy předchozí, bez ohledu na to, jestli je nějaký z počítačů *n*-násobně výkonnější (*n* je konstanta).

1.2 Asymptotická složitost

Asymptotická složitost je jeden z nejčastějších hodnotících kritérii algoritmů. Je to porovnání algoritmu s určitou funkcí pro *n* blížící se nekonečnu. Může nabývat tří různých složitostí.

O – Omikron (velké O, O, big O) – horní hranice chování

Ω – Omega – dolní hranice chování

Θ – Theta – vyjadřuje třídu chování

Tabulka 1: Tabulka nárůstu složitostí

Složitost	Počet operací – předpokládáme že jedna operace trvá 1 µs (10-6 sec)					
	10	20	40	60	500	1000
$\log_2 n$	3,3 µs	4,3 µs	5 µs	5,8 µs	9 µs	10 µs
n	10 µs	20 µs	40 µs	60 µs	0,5 ms	1 ms
$n \log_2 n$	33 µs	86 µs	0,2 ms	0,35 ms	4,5 ms	10 ms
n^2	0,1 ms	0,4 ms	1,6 ms	3,6 ms	0,25 s	1 s
n^3	1 ms	8 ms	64 ms	0,2 s	125 s	17 min
n^4	10 ms	160 ms	2,56 s	13 s	17 hod	11,6 dnů
2^n	1 ms	1 s	12,7 dnů	36000 let		
n!	3,6 s	77000 let	10^{34} let			

1.3 Výkonnost

Algoritmy, které jsou ekvivalentní se mohou lišit ve využívání výpočetních zdrojů. To lze označit jako kritická oblast v komplexních systémech, je to zejména možnost předvídat, jak se algoritmy budou chovat při jejich použití v rozsahu možných podmínek.

Jeden z nejdůležitějších zdrojů, jak dojít k určitému závěru je čas a využití paměti (potřebná paměť), které se mění v průběhu času s rostoucím množstvím dat. Program, který je

příliš pomalý, se pravděpodobně nebude využívat u aplikací ve kterých očekáváme real-time odpověď. Program, který využívá příliš mnoho paměti nemusí být dokonce proveditelný na standardních zařízení. I když využití paměti je v dnešní době poněkud zanedbatelný faktor, proto se klade největší důraz na časovou efektivnost algoritmů.

Časová efektivita algoritmu je měřena analýzou toho, jak se doba běhu mění s velikostí vstupních dat. Ve většině situací očekáváme že čas řešení daného algoritmu bude rostoucí s velikostí vstupních dat, či problému který má být řešen.

Příklad. ArrayList, za předpokladu že hledáme poslední prvek v poli a nepřistupujeme na konkrétní prvek přes index. Čím větší bude samotné vnitřní pole, tím déle nám bude trvat nalezení, iterování k poslednímu prvku.

Ovšem co je důležité, je měřit rychlost s jakou se doba chodu zvětšuje s velikostí vstupních dat. Například lineárně, exponenciálně nebo kvadraticky. Ovšem toto měřítko se spíše zaměřuje na vnitřní charakteristiku algoritmu než na vedlejší faktory, jako je rychlost počítače, na kterém běží samotný algoritmus, případně nějaké drobné změny a optimalizace kódu implementujícího algoritmus.

1.4 Složitost programu

Ve většině případů, se mnohdy upřednostňují jednoduché metody před více komplexními metodami. I za cenu toho že rozdělením jedné důmyslné metody na více menších a jednodušších metod, ztratíme na efektivnosti využití výpočetního času.

Samozřejmě jeden z důvodů je samotný čas programátora. Další fakt je také ten že žádný kód nikdy nezůstane čistě statický a je třeba jej upravovat. Programy jsou pravidelně upravovány a měněny, na základě požadavku na daný program, a to často jinými osobami než původním programátorem. Proto jednoduchý design a jednoduché algoritmy jsou velmi ceněny právě takovými osobami.

2 DATOVÉ STRUKTURY

Pod pojmem datové struktury si můžeme představit organizaci dat převážně v paměti, pro lepší efektivitu algoritmů, jako je fronta, zásobník, spojový seznam, halda, slovník a strom. Může obsahovat redundantní informace, například délka seznamu nebo počet uzlů v podstromu. Kapitola vycházela z (Pecinovský 2009, Roubalová 2015, Keogh a Davidson 2006, Prokop 2009, Knuth 2008)

2.1 ArrayList

ArrayList také často v programování označován jako dynamické pole. Je datový kontejner, struktura určená k uchovávání dat. Jak již napovídá název, dynamické pole, je struktura postavená nad polem (implementuje pole). Od statického pole se dynamické pole liší především že není omezeno pevnou velikostí jako tomu je u klasického pole. Dynamické chování ArrayListu je v tomto případě zastoupeno možností zvětšit své vlastní vnitřní pole. Ale toto dynamické chování je spojeno s jistou režií, která je v mnoha případech zbytečně drahá, například při známém počtu prvků, které vkládáme do kontejneru, takže jeho dynamičnost v konečném výsledku není využita.

Hlavní problém klasického pole je, potřeba vědět dopředu jeho velikost. V případě, kdy dojde k zaplnění celého pole prvky, nemůžeme již dále přidávat další. Stejně tak v případě odebírání, je spotřeba paměti stejná, bez ohledu na počet obsazených prvků v poli. Právě tyto nevýhody řeší dynamické pole, které si ukládá své prvky do vnitřního pole fixní délky a ve chvíli, kdy je kapacita vyčerpána, dojde k alokaci nového většího pole. Všechny prvky pole se překopírují a staré pole se smaže. Podobným způsobem se zachová, je-li hustota záznamu příliš řídká. Znovu dojde k alokaci, nyní již menšího pole a opět se do něj hodnoty překopírují. Tímto způsobem můžeme šetřit paměťové prostředky, na rozdíl od alokace nepřiměřeně velkého pole.

Stěžejní výhodou ArrayListu je jeho konstantní čas čtení prvků dle indexu, protože pole umožňuje náhodný přístup, proto můžeme konstatovat složitost O (1).

Ovšem jak již bylo zmíněno výše, na první pohled se postup při vkládání prvku nemusí zdát efektivní, zejména v okamžiku, kdy dojde k naplnění pole a ArrayList musí všechna data překopírovat. V tuto chvíli je asymptotická složitost vkládání prvku O(n). Díky tomu že k této expanzi nedochází často, můžeme tuto složitost rozložit v čase a tím se složitost vkládání amortizuje. Ve výsledku vyplyne, že amortizovaná složitost vkládání se blíží O (1). Je to v

důsledku možnosti zanedbat multiplikativní konstanty, a to jak v případě amortizované, tak asymptotické složitosti.

Ač tomu asymptotická složitost na první pohled nenapovídá, je díky amortizované složitosti struktura dynamického pole velmi efektivní. Ovšem si asymptotická složitost nese svá rizika, především nemožnost garantovat rychlost konkrétní operace, a proto není vhodné používat strukturu na RTOS (anglicky real-time operating system), kdy musíme garantovat určitou odezvu.

Případy vhodného použití ArrayListu:

- Není předem známý počet prvků,
- je-li vyžadováno indexový přístup k prvků,
- vkládání a odebírání probíhá pouze na konec struktury.

2.2 Vector

Datová struktura Vector je velmi podobná ArrayListu. Ale přesto má Vector dost podstatných odlišností od samotného ArrayListu. Ačkoliv v dnešní době již není Vector tolik využívaný a téměř ve všech případech byl nahrazen ArrayListem.

Největší odlišností od ArrayListu je především, to že Vector je jako zastánce datové struktury na poli synchronní. Docílením integrity dat je možné docílit stavu kdy, nemůžou dvě rozdílné aplikace zapisovat nebo mazat v jeden čas. A musí si vyčkat na uvolnění zdroje pro danou operaci.

Dalším znatelným rozdílem od ArrayListu je velikost realokace nového pole. Samotný Vector si alokuje dvojnásobek původní velikosti oproti ArrayListu, který si alokuje jen o jednu polovinu větší pole. Tímto způsobem sice omezíme režii na realokaci pole, ale bohužel nešetříme paměťové nároky na datovou strukturu, která je samozřejmě větší než u ArrayListu.

Stejnou nevýhodu dynamičnosti si nese i Vector, v případě odebírání prvků. Stále zůstává rezervované místo v paměti pro interní pole. A toto místo je uvolněno až v případě realokace na menší pole, které dochází pouze v případě kdy je Vector prázdný ze dvou třetin své původní velikosti.

Stejnou výhodu byla popsána v a, má také Vector. Výhoda se týká vkládání na konec seznamu, kdy je doba vložení konstantní. Ale ve chvíli, kdy chceme prvek vložit na začátek, pro nás znamená přerovnat celé pole o jeden prvek a tím uvolnit první místo pro vkládaný prvek, proto stejně jako ArrayList i zde je složitost vkládání O(n).

Stejně tak tomu je, v případě zaplnění celého alokovaného pole. Dojde k realokaci většího pole a překopírování veškerých prvků z prvního do druhého pole. A i zde je asymptotická doba složitosti O(n).

Samozřejmě lze tuto složitost rozložit v čase a tím se složitost vkládání amortizuje. Ve výsledku tak zjistíme, že amortizovaná složitost vkládání je O (1). Je to v důsledku ignorace multiplikativní konstanty, a to jak v případě amortizované, tak asymptotické složitosti.

Případy vhodného použití Vectoru

- Není předem známý počet prvků,
- je vyžadováno konstantní čtení prvků,
- vyžadujeme integritu dat,
- vkládání a odebírání probíhá pouze na konec struktury.

2.3 LinkedList

Spojový seznam (anglicky LinkedList) je jeden z nejběžnějších a nejjednodušších dynamických datových struktur. V podstatě existují čtyři základní typy spojitých seznamů. Jednosměrně spojité seznamy, dvojitě spojité seznamy, cyklicky jednosměrně spojité seznamy, cyklicky obousměrné spojité seznamy.

Obecně se záznam (uzel) skládá z režijní struktury a odkazu na nesená data. V jednosměrně zřetězeném seznamu obsahuje každý záznam, data a také odkaz na další prvek. Tímto způsobem můžeme jednosměrně procházet seznam ale pouze jedním směrem. V případě dvojitě zřetězeného seznamu každý prvek obsahuje navíc od jednosměrně zřetězeného seznamu, odkaz na prvek předchozí. Tím můžeme na rozdíl od jednosměrného seznamu, procházet seznam v obou směrech. Cyklicky jednosměrný zřetězený seznam má všechny vlastnosti stejnosměrného zřetězeného seznamu. A navíc poslední prvek má v sobě následující referenci na první prvek. Obousměrně zřetězený cyklický seznam má opět stejné vlastnosti jako necyklický obousměrný seznam s rozdílem, první prvek má jako svého předchůdce poslední prvek, poslední prvek má jako svého následníka první prvek.

Přední výhodou lineárního seznamu oproti konvenčnímu poli je vkládání na začátek, případně na konec seznamu, při této operaci má lineární seznam asymptotickou složitost O (1), bez nutnosti realokace nebo reorganizace celé datové struktury. Datové položky nemusí být ukládány souvisle do paměti nebo na disk. Tím mohou docílit konstantního přidávání prvků na začátek nebo konec seznamu. Dalšími výhodami je jednotková složitost v případě

sekvenčního prochází, nebo vkládání či odebírání dat sousedícího s konkrétním indexem, který je právě zpracováván.

Ovšem s výhodou přidávání a odebírání jsou spojeny jisté nevýhody. Takové lineární seznamy neposkytují rychlí přístup k indexovým datům nebo jakoukoliv účinnou formu indexování. Například při získání posledního uzlu v seznamu, za předpokladu že si neudržujeme poslední prvek jako samostatný odkaz, nebo nalezení daného uzlu případně umístění uzlu, který obsahuje námi hledaný prvek, může vyžadovat sekvenční skenování většiny nebo v nejhorším možném případě všech prvků, v závislosti na uložení uzlu. Tím se stává vyhledávání v lineárním seznamu lineární v čase. Tedy můžeme konstatovat, vyhledávání má složitost O (n).

Jakákoliv varianta lineárně zřetězeného seznamu není vhodná pro užití, v kterém očekáváme rychlé čtení prvků. Naopak, při častém vkládání a odebírání na začátek nebo konec seznamu se tato datová struktura velice hodí. Můžeme tím docílit téměř konstantního přidávání a odebírání. Ovšem samozřejmě za větší paměťové režie.

Případy vhodného použití jakékoliv varianty lineární seznamu

- Není předem známý počet prvků,
- předpokládá se sekvenční čtení.

2.4 Set

Datová struktura Set (v češtině množina) je abstraktní datová struktura, která obsahuje hodnoty, aniž by nám garantovala pořadí prvků. Jedná se o matematickou implementaci termínu množina, z tohoto důvodu obsahuje vždy každý prvek pouze jednou.

Set je datová struktura reprezentující rozhraní kolekcí. Samotná datová struktura je z části podobná kolekci List. Ovšem největší předností kolekce Set je že samotná struktura nedovoluje ukládání, resp. uchování, duplicitních hodnot. Samotná implementace kolekce Set není synchronní a musí se v případě potřeby synchronizovat externě. Ovšem Set není často synchronizován proto, že nebezpečné operace jsou přidávání a odebírání, které neprobíhají často. Operace přístupu může probíhat asynchronně.

Třída TreeSet implementuje rozhraní Set, které používá strom pro ukládání prvků. Objekty třídy TreeSet jsou uloženy ve vzestupném pořadí. Samotné operace jsou log (n) časově náročné. Jedná se právě o tyto operace, zápis, čtení a odstranění. Stejně jako HashSet je i TreeSet kolekcí která obsahuje jedinečné prvky. TreeSet jako takový není synchronní, proto se může v případě potřeby synchronizovat externě.

Mezi své výhody:

- Udržuje vzestupné pořadí,
- přístupové a vyhledávací časy jsou rychlé.

Třída HashSet implementuje rozhraní Set. Jak již název napovídá, jeho implementace je založena na Hash tabulce, která je instancí HashMapy. Samotná implementace na rozdíl od TreeSet neposkytuje vůbec žádnou záruku konstantního pořadí prvků v průběhu času (tzn. pořadí prvků, se v průběhu času mění). Duplicitní hodnoty nejsou v této třídě povolené, ale zároveň tato implementace dovoluje vložení nulového prvku.

Třída nabízí konstantní časový výkon pro základní operace jako je vkládání, odebírání a čtení dle klíče. Samozřejmě za předpokladu že hash funkce řádně rozptýlí prvky mezi větvemi. Samotné objekty jsou vloženy na základě jejich hash kódu.

Samotná iterace skrz tuto třídu je úměrná součtu velikosti instance HashSet (počet prvků) a „kapacitě" (počet větví). Proto je velmi důležité, aby nebyla nastavena počáteční kapacita příliš vysoká (nebo aby nebylo příliš nízké zatížení), pokud je pro nás důležitý iterační výkon.

2.5 Map

Přechozí kolekce byly založené na seznamu či poli. Jeden z dalších druhů kolekcí jsou kontejnery, které implementují rozhraní Map. V české terminologii se často také používá označení tabulka. Tyto kolekce ukládají elementy jako klíč-hodnota páry. Rozhraní datové struktury Map, nedovoluje jakékoliv duplicitní klíče a každý klíč může mapovat pouze jednu hodnotu.

Rozhraní Map nám nabízí tři pohledy, které nám umožní, aby byl obsah Mapy zobrazen jako sada klíčů, kolekce hodnot, nebo jako sada mapující klíč-hodnotu. Pořadí Mapy je definováno jako pořadí, v němž iterátory na mapě sbírají své prvky.

Například třída jako je TreeMap nám dává jasné záruky pořadí prvků, které vkládáme, naopak třída HashMap nám nedává absolutně žádné záruky, co se týče pořadí vložených prvků.

TreeMap implementuje rozhraní Map. Samotná implementace je založena na Red-black tree. Mapa je tříděná podle přirozeného uspořádání jejich klíčů nebo případě komparátoru, který musí být poskytnut při vytváření mapy v závislosti na tom, který konstruktor je použit. Tato implementace garantuje časové náklady pro operace čtení, vložení a odstranění log (n).

Zároveň tato implementace není synchronní. Proto se musí tato mapa v případě potřeby externě synchronizovat.

HashMap je jedním z dalších způsobů implementace rozhraní Map. Tato implementace dovoluje využívat veškeré funkcionality datové struktury Map a zároveň dovoluje vložení nulové hodnoty a nulového klíče.

Třída HasMap je velmi podobná datové struktuře HashTable, kromě toho že je nesynchronní a dovoluje vkládání *null*. Zároveň neposkytuje žádnou záruku, co se týče pořadí prvků, zejména nezaručuje, že zůstane pořadí prvků konzistentní v průběhu času.

Tato implementace zajišťuje konstantní výkon pro základní operace čtení a zápis O (1), za předpokladu že hash funkce vhodně rozděluje prvky mezi své větve.

Iterace přes celou datovou strukturu HashMap vyžaduje čas, úměrný kapacitě instance HashMap (počet větví) plus její velikost (počet mapování klíč-hodnota) Proto je velmi důležité, aby nebyla nastavena počáteční kapacita příliš vysoká nebo aby nebylo příliš nízké zatížení, pokud chceme dosáhnout co pokud možno největšímu iteračnímu výkonu.

Samotná instance HashMap má dva parametry, které ovlivňují její výkon. A tou je počáteční kapacita a faktor zatížení. Kapacita je počet „větví“ v tabulce hash a počáteční kapacita je jednoduše kapacita v okamžiku vytvoření tabulky hash. Faktor zatížení je měřítkem toho, jak je zaplněná tabulka hash před jejím zvětšením. Když počet záznamů v tabulce hash překročí hodnotu faktoru zatížení a aktuální kapacitu, tabulka hash je rehashed (tj. interní datové struktury jsou přestavěny) tak, aby hash tabulka měla přibližně dvojnásobek počtu větví.

2.6 Observable

Observable varianty datových struktur se skládají ze statických metod, které jsou 1:1 kopie metod implementovaných v kolekcí. Samotná implementace datové struktury se téměř neliší od obyčejné kolekce. Hlavním rozdílem, je možnost přidání posluchače, který naslouchá, a vyvolává událost v případě, že v kolekci došlo ke změně. Při změně je každý zaregistrovaný posluchač upozorněn. Množství posluchačů není nijak omezeno.

V podstatě je rozšířena původní kolekce, která je oblena její Observable variantou. Následně v případě přidání prvku do listu, posluchač obdrží oznámení vždy, když je provedena změna. Zároveň všechny metody jsou optimalizovány tak, aby poskytovaly pouze omezený počet oznámení. Tím můžeme předejít případnému zahlcení posluchače oznámeními o změně, ale také předejdeme tomu abychom příliš neztráceli čas pro obsluhovače.

3 POUŽITÉ NÁSTROJE A PROSTŘEDÍ

Pro potřeby uskutečnění experimentů je uvedeno nastavení prostředí, ve kterém probíhaly. V této kapitole je také popis dalšího použitého softwaru.

3.1 Použité prostředí

Při práci byl využit operační systém Microsoft Windows 10 Home v1803. Jedná se o 64b operační systém. Použitý operační systém je vázán OEM licencí. Taková to verze se váže k danému počítači, konkrétně k základní desce.

Pro testování byla zvolena Java 8 s aktualizací 191, vydanou k datumu 16. října 2018, pod licencí GNU General Public License (GPL).

IntelliJ IDEA je inteligentní vývojové prostředí neboli IDE (Integrated Development Environment, česky integrované vývojové prostředí) pro vývoj v jazyce Java ale také v mnoha dalších programovacích jazycích. Toto vývojové prostředí je zaměřeno na produktivitu vývojářů, díky možnosti zlepšení kódu využitím mnoha refaktorovacích nástrojů a funkcí, ať již se jedná o integrované nástroje nebo nástroje a moduly třetích stran. Velkou předností je velmi dobrá spolupráce s verzovacími systémy, plná klávesnicová podpora. Pro účely testování byla využita studentská licence, která je volně dostupná na oficiálním webu IntelliJ IDEA, lze snadno získat díky studentskému emailu, alternativou je možné zadat platné číslo studentského průkazu ISIC.

Pro zpracování bakalářské práce byl použit Microsoft Word a následně také pro vykreslování grafů Microsoft Excel. Oba nástroje byly licencované v rámci Office 365 licenčních programů Microsoft Office, konkrétní licence Office 365 A1 Plus for students, které nabízela Univerzita Pardubice všem svým studentům po dobu studia zdarma. Nástroje byly nainstalované bez jakéhokoliv dodatečného softwaru

3.2 Použitý hardware

Část bakalářské práce, převážně textová byla zpracována na notebooku HP EliteBook 840 G3, bližší specifikace nejsou třeba specifikovat. Tento notebook byl ovšem čase obměněn za novější HP EliteBook 1040 G4, jehož specifikace také nejsou třeba specifikovat, protože nezasahovaly do testování.

Testování datových struktur probíhalo na notebooku zakoupeném jako celek z následujících součástí. Komponenty v sestavě notebooku: Základní deska Lenovo 5B20F78873 TOUCH Y50-70 LA-B111P, procesor Intel® Core™ i7-4720HQ (Haswell architektura, Cache – 6 MB, Clock Rate 2600 – 3600 MHz, 4 jádra-8 vláken, 64 bitová

podpora), dále 16 GB DDR3, 1600 MHz (1,6 GHz) běžící na Dual channelu. Grafickými kartami jsou Intel® HD Graphics 4600 s 2 GB paměti a NVIDIA GeForce GTX 960M s 4 GB paměti, přesnější specifikace grafické karty není nezbytné znát. Pevný disk je Samsung SATA3 256 GB (model MZ7LN256HCHP-000L7) - 6 Gb/s.

Po startu operačního systému bylo k dispozici 13,7 GB volné operační paměti, operační systém si tedy pro svůj běh bral 2,3 GB RAM. Po spuštění vývojového prostředí, se spotřeba RAM v klidovém režimu zvedla na 3,3 GB. Během celého testování bylo povoleno swapování na disk. Nastavení vývojového prostředí pro build a následný běh aplikace bylo na dvou vláknech. Pro potřebu testování nebyla vyhrazována paměť RAM.

Sledování prostředků probíhalo za pomoci aplikace Sledování prostředků, která byla zabudována do operačního systému. Velký důraz se kladl na sledování zásahů různých programů, například antivirového programu.

4 METODIKA SROVNÁNÍ ALGORITMŮ

Pro vlastní měření bylo využito aplikace teorie velkých čísel a hrubé síly za předpokladu stability dat. Každý testovaný experiment se opakoval v rozmezí jedno sto tisíckrát až jeden milion krát. Tím bylo docíleno vysoký počet opakování, že výsledky, které se podařilo naměřit, se velmi blíží skutečným hodnotám.

Princip metodiky měření spočívá v testování struktur či jejich algoritmů různá *n* (*n* je počet vstupů). Každý z testů byl vždy replikován *k*-krát.

Podle speciálního algoritmu se generuje počet prvků na vstupu. Tento algoritmus se stará o to, aby nebylo příliš velké množství zaznamenaných testování, ale také aby data byla škálována dle patřičné míry. To znamená, zamezení příliš hustému, nebo v opačném případě, příliš řídkému rozložení dat. S co největší jednoduchostí, se kterou pracuje algoritmus lze dosáhnout minimální režie.

Stabilitou je míněno, zda je zachováno pořadí prvků. Protože pořadí prvků je jeden z nejdůležitějších parametrů, které by mohly značně ovlivnit testování, bylo rozhodnuto, že pořadí prvků zůstane vždy konzistentní. V případě, kdy by nebylo dodrženo pořadí prvků, bylo by možné upravit algoritmy tak, aby i přidávání na začátek, například v ArrayListu, dosáhlo složitosti O (1), stačilo by pouze prvek na daném indexu přesunout na konec pole a na vzniklé místo vložit vkládaný prvek. Ovšem došlo by ke ztrátě pořadí, proto od takové implementace bylo upuštěno. Kapitola vycházela z (Everitt 2002, Lewis a Denenberg 1991, Aho et al. 1983)

4.1 Výběr struktur

Pro testování bylo zvoleno pět nejvyužívanějších datových struktur, a to pět struktur s vlastní implementací a ty samé nativně implementované struktury. Každá z těchto struktur byla podrobena testu. Tento test se zaměřoval na konkrétní časovou složitost daných operací jako je přidávání prvků, odebírání, hledání dle indexu, hledání dle klíče.

Zkoumaným parametrem byl průměrný čas na provedení jedné operace v rozsahu *n* prvků. Testy byly provedeny jak pro datový typ Integer, tak také pro datový typ String. Následně získané hodnoty byly zpracovány v tabulkovém procesoru a vizualizovány formou přehledného grafu.

Takto získané grafy byly porovnány mezi sebou, aby se dalo lépe nastínit, jaké mezi nimi byly rozdíly, či naopak, že jejich průběhy byly podobné. Porovnávala se každá nativní

datová struktura oproti strukturám s vlastní implementací. Následně byl vytvořen přehled zkoumaných struktur, převážně struktur vlastní implementace.

4.2 Programovací paradigmata

Jak již bylo zmíněno výše, pro vlastní měření bylo využito hrubé síly (v angličtině Brute-Force). Tento způsob patří mezi jedny z nejtriviálnějších způsobů řešení daného problému.

Brute-Force algoritmus pracuje tak, že v případě, kdy známe způsob, jak docílit výsledku, nebereme ohledy na optimalizaci výpočetního procesu a pouze díky výpočetní síle stroje dojdeme k výsledku. Samotné algoritmy tohoto typu jsou jednoduché na implementaci a mnohdy mají malou náročnost na paměť, kterou využívají pro vlastní režii. Ovšem hlavní nevýhodou takovýchto algoritmů, při vzrůstajícím počtu zpracovaných prvků *n*, extrémně roste jejich výpočetní složitost.

Ačkoliv je tento způsob řešení velmi náročný na výpočetní složitost, je to mnohdy jediný způsob, jak dojít k výsledku. Zároveň je tento způsob mnohem efektivnější pro malé *n* oproti složitějším algoritmům. Tím jsou míněny algoritmy, které jsou náročnější na paměťové nároky nebo výpočetní výkon, který spotřebovávají na vlastní režii.

Jedním z nejdůležitějších faktorů při implementaci je, zdali máme strukturu, která zachovává stejné pořadí prvků. V některých případech, zle implementovat takovou sktrukturu u které není důležité utříděnost prvků ve struktuře. V takovém případě lze přizpůsobit implementaci. Například u Arraylistu, při vkládání na konkrétní index, vložit prvek na konec seznamu a pouze ho prohodit s indexem.

Tabulka 2: Vliv utříděnosti na složitost

		Vkládání	Přístup	Odebírání
ArrayList	Utříděná	O (n)	O (1)	O (n)
	Netříděná	O (1)	O (n)	O (n)
Vector	Utříděná	O (n)	O (1)	O (n)
	Neutříděná	O (1)	O (n)	O (n)
LinkedList	Utříděná / neutříděná	O (1)	O (n)	O (1)
TreeMap		log (n)	log (n)	log (n)
TreeSet		log (n)	log (n)	log (n)

V tabulce 3 lze vidět kompletní přehled testovaných operací pro konkrétní datové struktury.

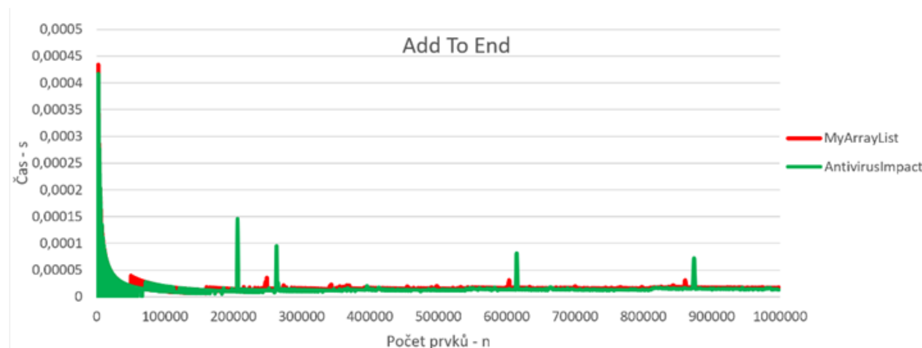
Tabulka 3: Stav řešení

			Vkládání				Přístup				Odebírání			
			1	n	ran	nex	1	n	ran	key	1	n	ran	key
ADT Pole	ArrayList	vlastní	X	X	X				X			X	X	
	Vector	vlastní	X	X	X				X			X	X	
	ArrayList	Java	X	X	X				X			X	X	
	Vector	Java	X	X	X				X			X	X	
ADT Seznam	Singly	vlastní	X	X	X				X		X	X		
	Doubly	vlastní	X	X	X				X		X	X		
	Circular	vlastní	X	X	X				X		X	X		
	Multiply	vlastní	X	X	X				X		X	X		
	LinkedList	Java	X	X	X				X		X	X		
Stromy	TreeMap	vlastní				X				X				X
	TreeMap	Java				X				X				X
	TreeSet	vlastní				X				X				X
	TreeSet	Java				X				X				X

4.3 Testovací sestava

Před započetím samotné procesu testování, bylo nutné sledovat procesy různého softwaru, ať už se jednalo o procesy běžící na pozadí, nastavení služeb či zásahy antivirového programu. Nutnost zajistit plnohodnotné a neměnné prostředí pro spuštění testů bylo prioritou, která jak se v následujícím grafu ukázala, nebyla zanedbatelná.

Na obrázku 2 je graf ve kterém byly pozorovány zásahy antiviru, který při operaci s pamětí významně zvyšoval režii. Pro srovnání byl vložen i tentýž výstup testu provedený po odstranění antiviru. Je zřejmé, že režie na operaci s pamětí tu stále je, nicméně již není tak vysoká, jako za přítomnosti antivirového software.



Obrázek 2: Režie antivirového programu

Za těchto neměnných podmínek můžeme konstatovat, že bylo dosaženo typového stroje. Za takový stroj lze označit počítač, který má stejný hardware a stále stejný software. Dosažením ekvivalentních podmínek pro testy lze prohlásit, že každá struktura měla rovnocenné podmínky.

Princip použití testovací soustavy lze zachytit v několika bodech. Tyto body byly následně použity jako vodítko pro následnou implementaci. Samotný experiment, náhodný výběr z dynamického pole, probíhal v testovacím prostředí, a to v konkrétních krocích.

1. Z každého kroku cyklu se určí šířka testu n, pro kterou bude realizována sada testů.
2.
    a. Spustí se cyklus, který vygeneruje pro šířku n dostatečné množství hodnot.
    b. Spustí se testování první metody pro šířku n. Tento test se opakuje tolikrát, aby došlo k požadovanému počtu opakování.
    c. Uloží se čas do proměnné z bodu 2b.
    d. Spustí se korekční cyklus, který určí dobu, po kterou byly v bodě 2b prováděny režijní operace.
    e. Uloží se finální čas z bodu 2b – čas z bodu 2d.
    f. Dojde k zápisu řádku (informace o hodnotě n, a naměřený čas)
3. Opakuje se bod 2 dokud není dosaženo požadované šířky.
4. Experiment se zapíše do souboru.
5. Ukončení experimentu.

Během běhu testu, nebylo s počítačem nijak manipulováno. Provedení experimentu se časově velice liší dle složitosti dané operace. Časy které, byly naměřeny, se pohybují od jedné minuty až do více než 36 hodin pro jeden typ testu. Proto bylo v některých případech

upuštěno od testování v celé šířce, avšak bylo dodrženo rozmezí šířky testů od 100 000 až do 1 000 000. Veškeré výsledky dostupné v souboru .xlsx, připravené k dalšímu zpracování, budou přiloženy na datovém nosiči.

Protože naměřený čas nebylo možné přímo zpracovávat v testovacím programu, bez případné ztráty přesnosti muselo být takto činěno v tabulkovém procesoru. Jednalo se pouze o jednoduché matematické operace, nicméně tento postup byl nezbytný k zachování přesnosti, lepší manipulaci dat a pro následnou vizualizaci dat.

4.4 Stanovení předpokladů

a. Struktury založené na dynamickém poli za předpokladu téměř totožné implementace, budou mít velmi podobné časové průběhy. Stejně tomu bude u lineárních datových struktur, nebo struktury založené na stromové struktuře.
b. Struktury založené na lineárním seznamu budou rychlejší v zápise než struktury založené na dynamickém poli
c. Přístup u dynamických datových struktur bude extrémně rychlý
d. Vlastní datové struktury budou mít stejné, nebo lepší výsledky.
e. Operace zpřístupnění u datové struktury založené na stromu, bude dramaticky rychlejší než implementace lineární datové struktury.
f. Observable varianty struktur budou mít velmi podobné výsledky jako obyčejné struktury.

4.5 Implementace

Pro potřeby testování bylo nutné vytvořit si vlastní testovací aplikaci. Tato aplikace je jako samotný program implementovaný v jazyce Java. Bylo zvoleno řešení pouze konzolovou aplikací (aby byla minimalizována režie a zároveň se snížil dopad běhu programu na test, z tohoto důvodu bylo upuštěno od grafického prostředí).

Obrázek 3 nabízí pohled na výstupní konzoli testovacího programu. Lze vidět že bylo provedeno několik druhů testů, testy s variantou pro datový typ Integer a String. Pro datovou strukturu TreeMap záměrně nebyl vypsán datový typ. Protože datová složka se skládá z klíč-hodnota páru, kde jako klíč byl využit typ Integer a jako hodnota byl použit String.

```
        Metod    Operation      Structure      Type     summary
Test   random      get          ArrayList    (Integer)   done!
Test   random      get          ArrayList    (String)    done!
Test   toEnd       add          ArrayList    (Integer)   done!
Test   toEnd       add          ArrayList    (String)    done!
Test   fromEnd     remove       ArrayList    (Integer)   done!
Test   fromEnd     remove       ArrayList    (String)    done!
Test   random      get          LinkedList   (Integer)   done!
Test   random      get          LinkedList   (String)    done!
Test   toEnd       add          LinkedList   (Integer)   done!
Test   toEnd       add          LinkedList   (String)    done!
Test   fromEnd     remove       LinkedList   (Integer)   done!
Test   fromEnd     remove       LinkedList   (String)    done!
        Method    Operation     Structure               Summary
Test    random      get          TreeMap                 done!
Test    toEnd       add          TreeMap                 done!
Test    random      remove       TreeMap                 done!
```

Obrázek 3: Ukázka výstupu konzole

Na obrázku 4 lze vidět organizaci provedených testů, bylo nezbytné testy členit do logických bloků pro přehlednost. Testy byly členěny dle své datové struktury, následně dle operace, kterou vykonávají.

```
public static void main(String[] args) {

    // test for LinkedList
        Add to begin test
        Add to end test
        Random add test
        Random get test

    // test for ArrayList
        Add to begin test
        Random get test
        Random add test


    // test for Vector
        Add to begin test
        Random get test

    // test for Map
        Add to Map
        Delete test
        Get test

    // test for Set
        Add to Set
        Delete test
        Get test
```

Obrázek 4: Ukázka základního členění logiky testů v programu

Obrázek 5 nabízí pohled na členění po logických blocích, ve kterých dále byly volány pouze metody, které prováděli konkrétní výpočet pro danou operaci.

```
// test for LinkedList
   //<editor-fold defaultstate="collapsed" desc="Add to begin test">
       testAddToBeginCircularLinkedList();
       testAddToBeginDoublyLinkedList();
       testAddToBeginMultiplyLinkedList();
       testAddToBeginSingledLinkedList();
   // </editor-fold>
   // <editor-fold defaultstate="collapsed" desc="Add to end test">
       testAddToEndCircularLinkedList();
       testAddToEndDoublyLinkedList();
       testAddToEndMultiplyLinkedList();
       testAddToEndSingledLinkedList();
   // </editor-fold>
   // <editor-fold defaultstate="collapsed" desc="Random add test">
       testRandomAddSinglyList();
       testRandomAddMultiplyList();
       testRandomAddDoublyLinked();
       testRandomAddCircularList();
       testRandomAddLinkedList();
   // </editor-fold>
```

Obrázek 5: Příklad volání metod testů v rozdělených do logických celků

Na obrázku 6 je konkrétní řešení výpočtu času při operaci náhodný výběr z datové struktury ArrayList. Implementace byla provedena na základě testovací soustavy a předem sestaveného postupu. Jak je na první pohled patrné, test byl proveden pro šířku jeden tisíc testů. Z každého takového experimentu byl spočítán velikost pole, pro které bude test prováděn.

Při generování náhodných hodnot pro náhodné zpřístupnění, bylo nutné pro ověřitelnost testu, generovat hodnoty dle násady. Násada je vždy získána z kroku cyklu. Vytvořením náhodné posloupnosti na základě násady, dostaneme pro každý takový cyklus stejnou posloupnost pseudo náhodných čísel. Stejná řada pseudo náhodných čísel nám umožňuje ověřit, jak se zadaný experiment chová pro různé implementace. Zadaná implementace se projevovala v rámci experimentu tak, že například v případě procházení pole, vždy vyhledáván prvek, který byl vždy stejně vzdálený od prvního prvku. Tímto způsobem bylo možné sledovat, jak rozdílné implementace jsou časově náročné.

```java
for(int experiment = 1; experiment<1000;experiment++) {
    Random r = new Random(experiment);
    int min = 0;
    int max = experiment * experiment;
    // prepare random numbers for experiment
    int[] arrayRanVar = new int[experiment * experiment];
    for (int rand = 0; rand < experiment * experiment; rand++) {
        arrayRanVar[rand] = (r.nextInt((max - min)) + min);
    }
    array = new ArrayList<String>();
    // prepare array for search experiment
    for (int a = 0; a < experiment * experiment; a++) {
        array.add("5");
    }
    // start experiment
    timeStart = Calendar.getInstance().getTimeInMillis();
    for (int i = 0; i < (experiment * experiment); i++) {
        array.get(arrayRanVar[i]);
    }
    timeStop = Calendar.getInstance().getTimeInMillis();

    // total time with overhead of program
    long endTime = ((timeStop - timeStart));
    // calculation overhead of program
    int randomNumberGetter;
    long randomtimeStart = Calendar.getInstance().getTimeInMillis();
    for (int e = 0; e < experiment * experiment; e++) {
        randomNumberGetter = arrayRanVar[e];
    }
    long randomTimeStop = Calendar.getInstance().getTimeInMillis();
    // final calculation running time of experiment
    endTime = endTime - (randomtimeStart - randomTimeStop);
    // save data to list
}
```

Obrázek 6: Náhled na konkrétní implementaci testu pro jednu operaci

5 VÝSLEDKY ANALÝZY

Výsledky analýzy byly velmi překvapivé. Některé naměřené hodnoty potvrdily fakt, že je možné naprogramovat stejně rychlou datovou strukturu jako je nativně implementována v Javě. Tento fakt se podařilo potvrdit u všech implementovaných struktur. V některých případech se dokonce potvrdilo, že vlastní naprogramované datové struktury jsou dokonce rychlejší než nativně implementované.

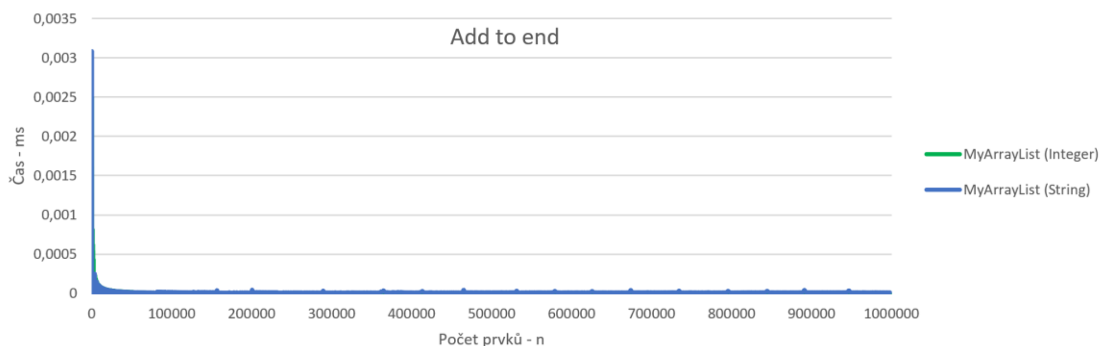
Vzhledem k velmi podobné implementaci datové struktury ArrayList a Vector, kdy bylo dosaženo stejných výsledků pro obě datové struktury. Bylo upuštěno od další testování, protože v každém případě bylo dosaženo stejných hodnot.

Podobně tomu bylo i v případě různých datových typů. Testy pro datový typ String a Integer vždy vycházeli stejně, nebo jen s opravdu velmi malými rozdíly, které nebyly vidět na vizualizaci grafu. Z tohoto důvodu bylo upuštěno od dalšího testování s různými datovými typy, neboť pozbylo smyslu.

5.1 ArrayList

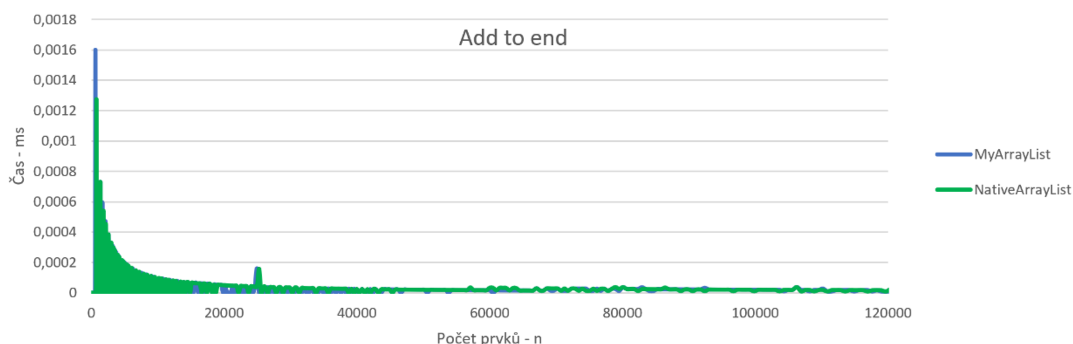
Obrázek 7: ArrayList vkládání na konec seznamu

Obrázek 7 znázorňuje graf průměrného času, který je potřebný na vložení prvku. S vyšší počáteční režií, která je následně amortizována.



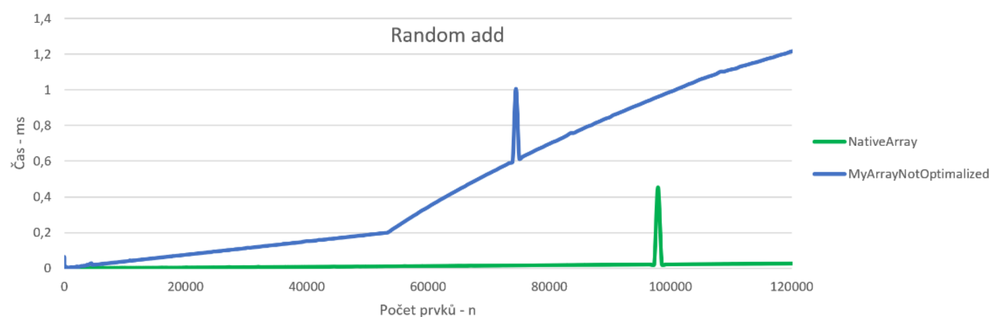
Obrázek 8: ArrayList rozdíl mezi datovými typy

Na obrázku 8 je graf, který znázorňuje rozdíly mezi datovými typy. Tento rozdíl byl zanedbatelný, proto autor rozhodl že další testování s různými datovými typy nebude prováděno. Stejné výsledky se projevili u všech testovaných datových struktur.



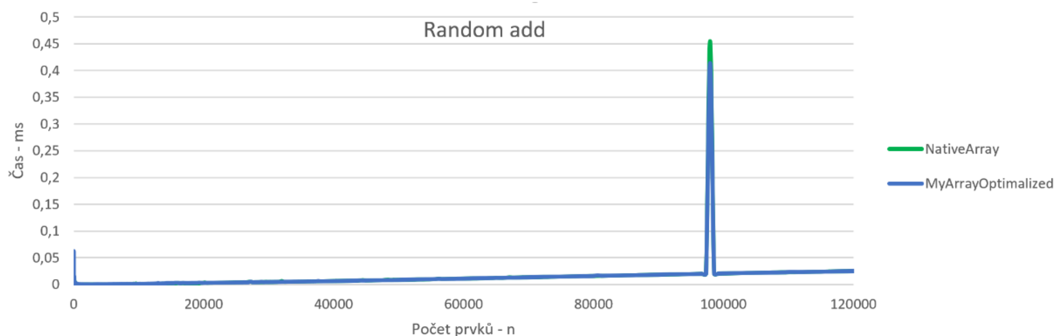
Obrázek 9: ArrayList porovnání s nativním ArrayListem

Na obrázku 9 je porovnání vlastní datové struktury s nativní strukturou, díky kterému můžeme prokazatelně tvrdit, že lze naprogramovat takovou datovou strukturu, která se vyrovná rychlosti nativně implementované struktuře ArrayListu, nebo ji dokonce předčí.



Obrázek 10: ArrayList náhodné vložení prvku do pole

Na obrázku 10 jsou vidět rozdílné časové průběhy náhodného vkládání do datové struktury, vlastní implementace a nativně implementované datové struktury. Tento rozdíl byl opravdu znatelný.



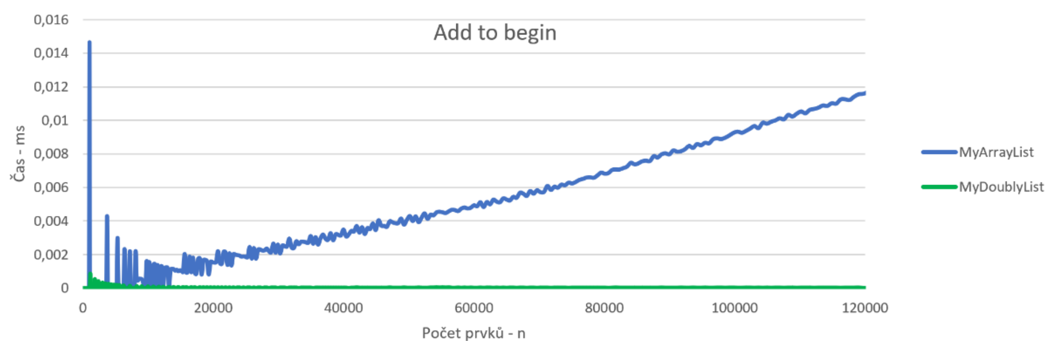
Obrázek 11: ArrayList porovnání nativní implementace s vlastní optimalizovanou

Na obrázku 11 je časový průběh již optimalizované datové struktury společně v porovnání s nativní datovou strukturou. Této optimalizace bylo dosaženo změnou v přerovnávání pole, a dokonce se povedlo dosáhnout stejných časů pro obě struktury.

```java
@Override
public void add(int index, E data) {
    if (size() < index) {
        throw new IndexOutOfBoundsException("You are out of index!!");
    } else if (index == size()) {
        add(data);
    } else if(size()==array.length){
        grow();
        add(index,data);
    } else {
        for (int i = size(); i > index; i--) {
            array[i] = array[i - 1];
        }
        numberOfElements++;
        array[index] = data;
    }
}
```

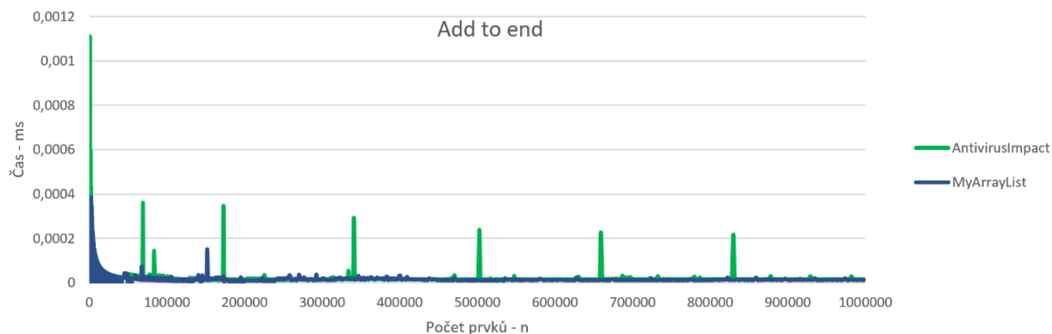
Obrázek 12: Stav metody před optimalizací

Na obrázku 12 lze vidět implementaci neoptimalizované metody vložení prvku na libovolné místo v ArrayListu. Lze vidět cyklus který způsobil zajímavý projev grafu na obrázku 10.



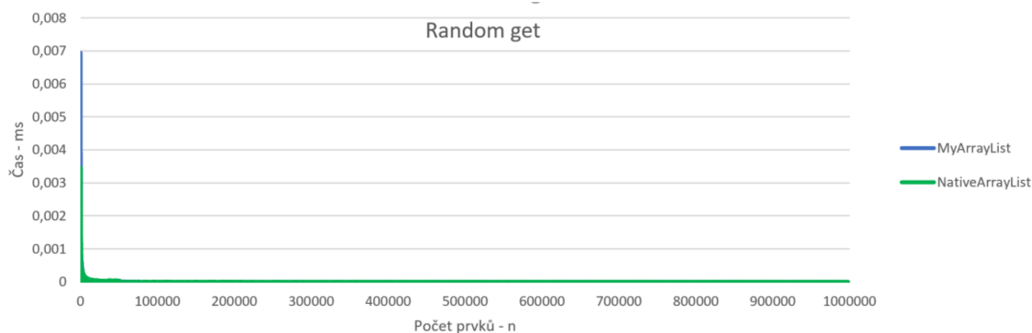
Obrázek 13: Časová amortizace vkládání na začátek ArrayListu

Na obrázku 13 je jedno z velmi zajímavých pozorování které bylo možno pozorovat, a to, jak se ArrayList s vkládáním na začátek chová. Počáteční vysoká režie při překopírování pole, následná amortizace času potřebného na kopírování, nicméně pořád byl graf vzestupné tendence. Pro představu byl přidán do grafu i záznam průběhu přidávání na začátek u spojového seznamu, konkrétní implementace je obousměrný spojový seznam.



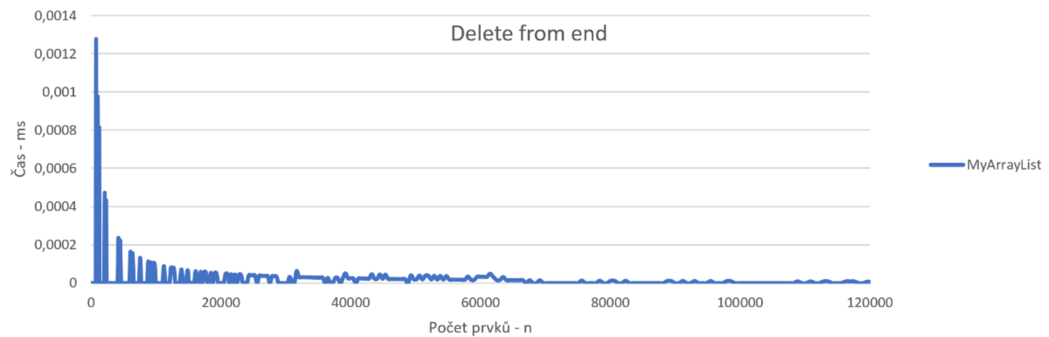
Obrázek 14: ArrayList vs. DoublyLinkedList metoda přidání na konec

Na obrázku 14 lze vidět zásahy antivirového programu, který při překopírování pole, vždy kontroloval obsah paměti. Tím se zvýšila časová režie, neboť bylo nutné pro další pokračování vyčkat na dokončení skenování antivirového softwaru. Proto bylo následně autorem rozhodnuto o odstranění antivirového programu a zopakování dalších testů, aby se potvrdil dopad antivirového programu.



Obrázek 15: Porovnání času k přístupu mezi nativním a vlastním polem

Na obrázku 15 zle pozorovat další důkaz, že bylo docíleno stejného výsledku jak pro nativně implementovaný ArrayList, tak i pro vlastní implementaci. Z tohoto důvodu další komentář není třeba.

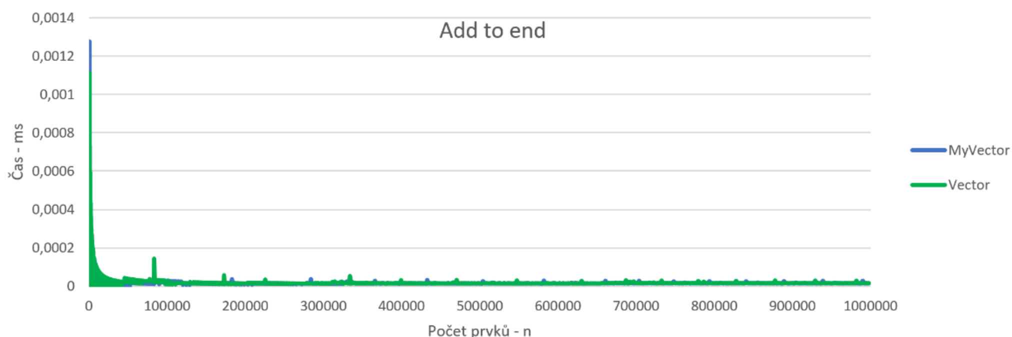


Obrázek 16: Průběh amortizace při odebírání z konce seznamu

Na obrázku 16 je příklad amortizace, ve kterém je v čase rozkládán potřebný čas na překopírování hodnot při zmenšování pole. Občasné excesy jsou způsobené spuštěním garbage colectoru.

5.2 Vector

Vzhledem k rozhodnutí autora implementovat datovou strukturu Vector na poli stejně jako tomu je u ArrayListu byly očekávány velmi podobné výsledky.



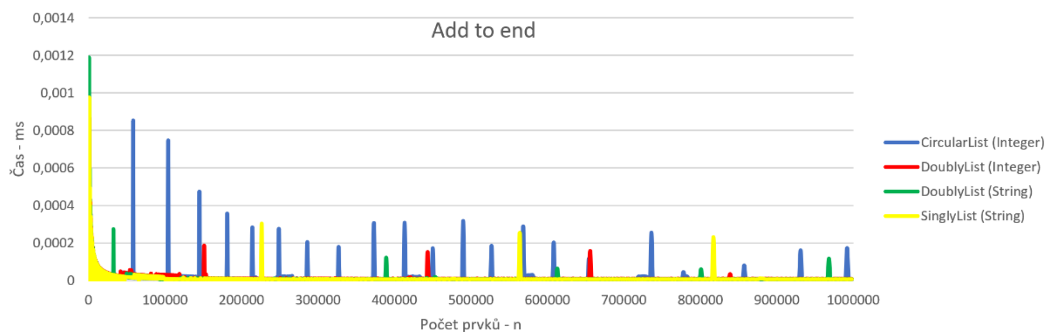
Obrázek 17: Porovnání vlastní implementace s nativní implementací Vectoru

Na obrázku 17 lze pozorovat výsledky chování struktury ArrayListu a Vectoru. Tyto výsledky ovšem byly velmi překvapivé, protože bylo zjištěno, že chování je totožné, jako tomu je u ArrayListu. Z tohoto důvodu se autor rozhodl, že pro potřeby této práce není nezbytné dále provádět testy na datové struktuře Vector.

5.3 LinkedList

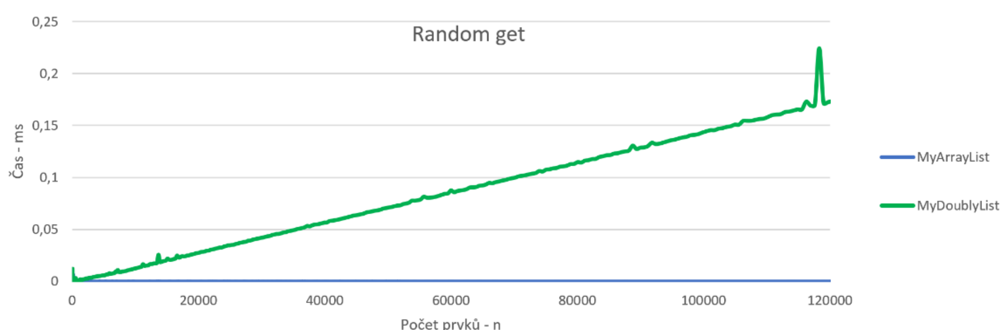
Datová struktura LinkedList má několik možných implementací. Všechny implementace byly popsané v teoretické části. Při testování čtyř implementací LinkedListu

bylo zjištěno, že všechny tyto implementace mají téměř stejný průběh svého chování. A tak autor na základě tohoto zjištění upustil od testování pro popsané implementace LinkedListu. Dále bylo pro testování vybrána implementace DoublyLinkedListu. Autor se takto rozhodl také na základě zjištění, že nativní datová struktura LinkedListu používá právě takovou implementaci.



Obrázek 18: Přehled implementací LinkedListu

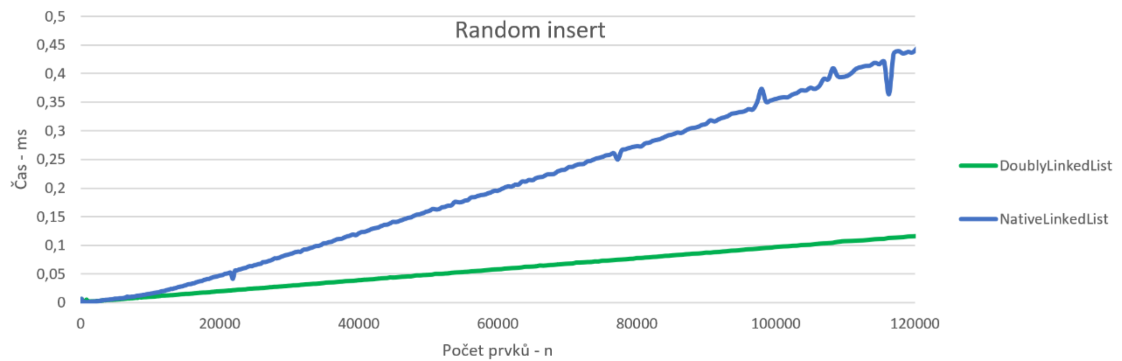
Obrázek 18 znázorňuje zápis na začátek seznamu s různými datovými typy. Jak zle snadno odvodit, datové typy nemají vliv na časovou náročnost dané operace. Proto v dalších testováních autor rozhodl vynechat testování pro různé datové typy. Excesy, které lze pozorovat, jsou zapříčiněné správou paměti, případně spuštěním garbage colectoru.



Obrázek 19: Náhodný přístup k prvku v seznamu oproti ArrayListu

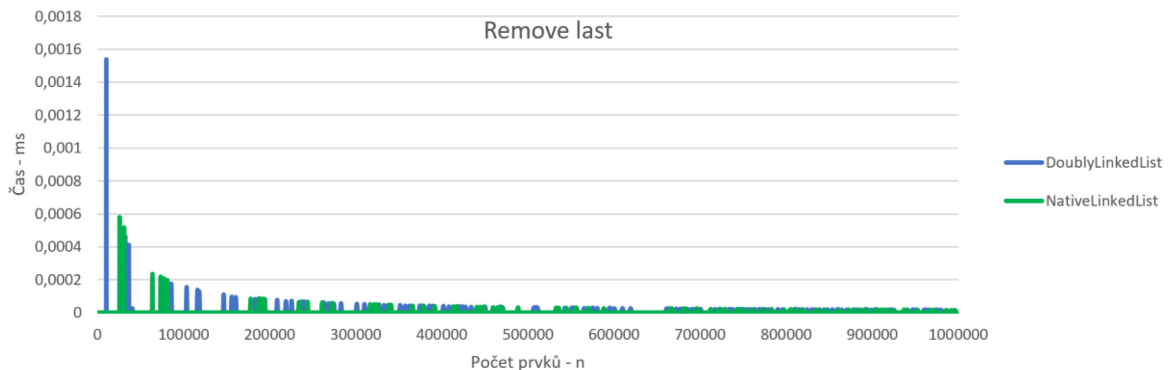
Na obrázku 19 lze pozorovat průběh náhodného přístupu u ArrayListu, vzhledem k možnosti spočítat si přímo adresu prvku je prakticky konstantní v čase. Bohužel LinkedList takovou možnost nemá, a tak je vidět že s počtem prvků, přes které musí cyklus proběhnout, aby nalezl hledaný prvek, se lineárně zvedá. V průběhu grafu je vidět, jak negativně ovlivnila

správa paměti časový průběh. Z části se o to zasloužila větší datová režie se kterou je spjatý seznam.



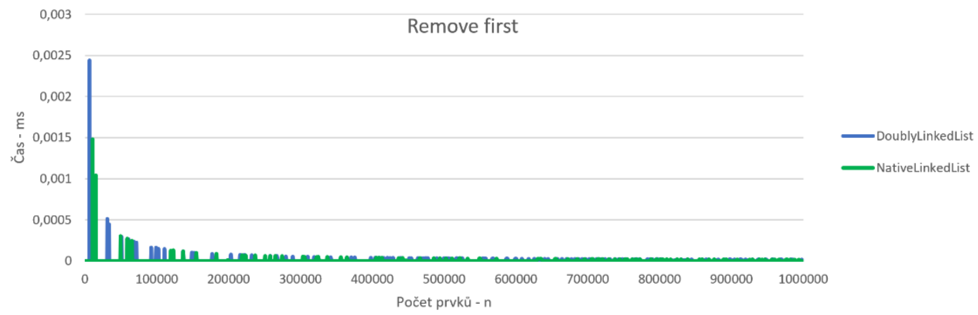
Obrázek 20: Vkládání na náhodnou pozici v seznamu

Na obrázku 20 lze vidět průběhy časů pro vkládání prvků do seznamu. Ačkoliv byly snahy o optimalizaci, nepodařilo se docílit stejné efektivity jako tomu bylo u ArrayListu.



Obrázek 21: Odebírání posledního prvku z LinkedListu vs. Nativní LinkedList

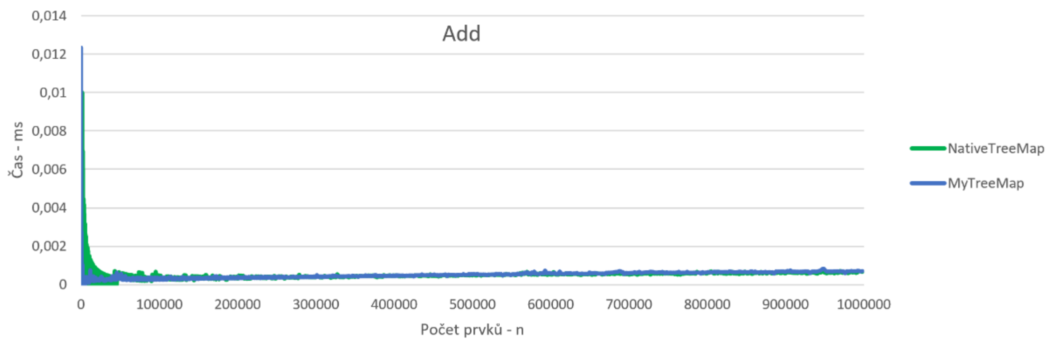
Na obrázku 21 je vidět časový průběh odebírání z konce seznamu. Seznam má referenci na poslední prvek v seznamu a zároveň implementace dovoluje obousměrné procházení, tudíž je možné se přesunout z posledního prvku na předposlední a tím docílit téměř konstantnímu odebírání.



Obrázek 22: Odebírání prvního prvku z LinkedListu vs. Nativní LinkedList

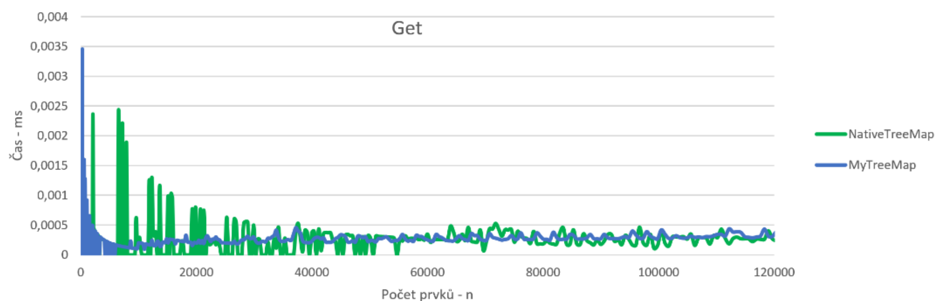
Obrázek 22 znázorňuje časový průběh odebírání ze začátku seznamu. Ten se liší od odebírání z konce seznamu pouze vyšší počáteční režií, která nejspíše byla způsobena garbage colectorem.

5.4 Map



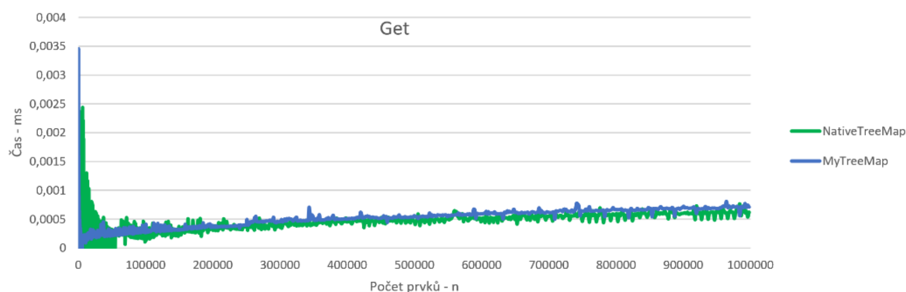
Obrázek 23: Operace přidání TreeMap vs. Nativní TreeMap

Na obrázku 23 je vidět časový průběh stromové struktury který je spotřebováván při operaci zápis. Vkládání probíhá do utříděné struktury.



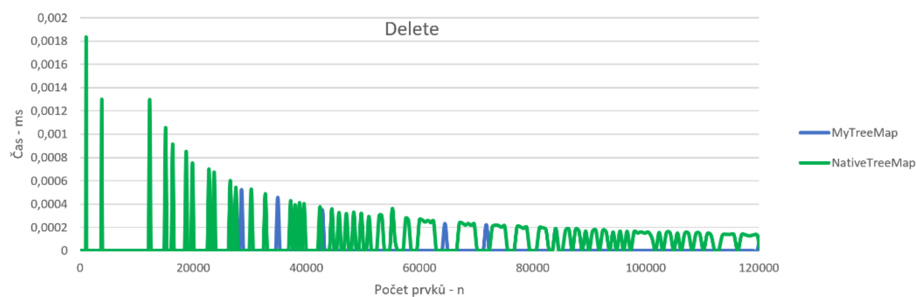
Obrázek 24: Operace čtení přiblížení začátku průběhu TreeMap vs. Nativní TreeMap

Obrázek 24 je krásným způsobem možno pozorovat, jak je vlastní implementace na počátku rychlejší než nativní implementace.



Obrázek 25: Průběh operace čtení v plné šířce TreeMap vs. Nativní TreeMap

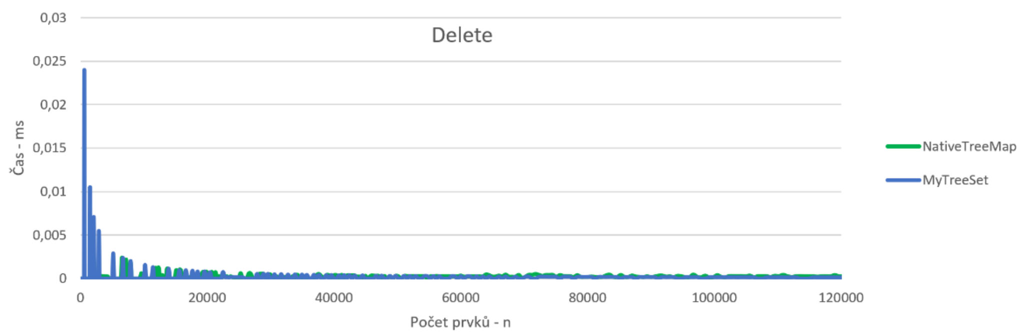
Na obrázku 25 je vidět, jak se z počátku rychlejší struktura pomalu převrací k vyšší časové náročnosti. Nicméně pořád jsou si časy velmi podobné.



Obrázek 26: Odebírání kořene TreeMap vs Native TreeMap

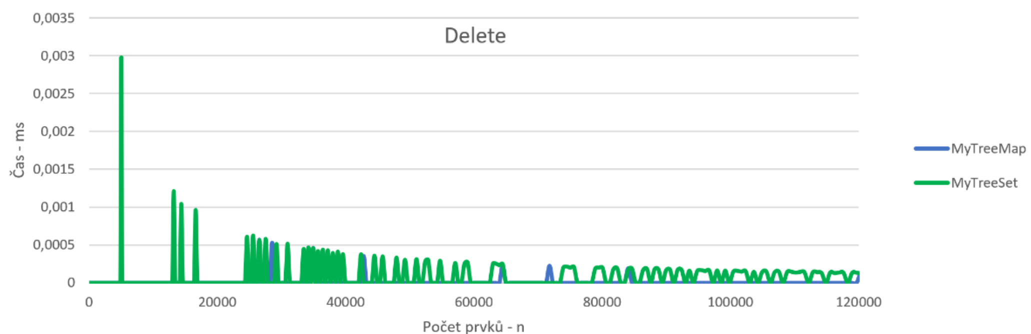
Obrázek 26 je krásným příkladem časové amortizace, zajímavým faktem je že se podařilo docílit o několik tisícin sekundy lepších časů, jak u nativní struktury.

5.5 Set



Obrázek 27: Odebírání kořene TreeSet vs Native TreeSet

Obrázek 27 zobrazuje průběh času při operaci odstranění mezi nativní strukturou a strukturou s vlastní implementací. Je vidět, že bylo docíleno téměř stejných výsledků.



Obrázek 28: Odebírání kořene MyTreeMap vs MyTreeSet

Obrázek 28 značí časový průběh mezi datovou strukturou TreeSet a TreeMap. Vzhledem k tomu že pro obě struktury byla použita ta samá implementace, není překvapivé, že i výsledky jsou pro obě struktury totožné. Z tohoto důvodu se autor rozhodl dále neprovádět testy pro tuto datovou strukturu.

5.6 Zhodnocení předpokladů

Předpoklad A

Testovány byly všechny datové struktury zmíněné v tabulce 3. Vzhledem k podobné, nebo totožné konstrukci datových struktur se opravdu prokázalo že struktury mají stejné časové průběhy. Tento předpoklad nevyvrátil ani pokus s různými datovými typy.

PŘEDPOKLAD POTVRZEN

Předpoklad B

Toto testování a vyhodnocení bylo velmi triviální. Jak již vyplynulo z teoretické části, cokoliv jiného, než potvrzení tohoto předpokladu by bylo krajně zavádějící. Předpoklad se opravdu prokázal, zle ho pozorovat na obrázku 13.

PŘEDPOKLAD POTVRZEN

Předpoklad C

Tento předpoklad byl založený na základě teoretické přípravy. Opět se prokázalo že se předpoklad splnil. Z velké části na tom záleželo právě z důvodu náhodného přístupu, které si nese pole pevné délky. Případně zle tento předpoklad pozorovat na obrázku 19 ve kterém je porovnání přístupu u ArrayListu a LinkedListu.

PŘEDPOKLAD POTVRZEN

Předpoklad D

Tento předpoklad se podařilo potvrdit pouze z části. Předpoklad se potvrdil u dynamických datových struktur a stromových struktur. Bohužel u lineárních struktur se tento předpoklad nepodařilo prokázat.

PŘEDPOKLAD ZAMÍTNUT

Předpoklad E

Tento předpoklad byl naznačen již v teoretické části. A dokonce byl dokázán prakticky. Stromová struktura byla opravdu mnohonásobně rychlejší než implementace lineární datové struktury. Tento předpoklad byl úspěšný především, kvůli sekvenčnímu procházení seznamové struktury.

PŘEDPOKLAD POTVRZEN

Předpoklad F

Opět další předpoklad, který byl potvrzený pouze z části. Hlavní nevýhoda observable varianty je v tom, že musí upozorňovat posluchače, které byly k struktuře připojeny. A právě v závislosti na tom, kolik je připojených posluchačů, se odvíjí časové zdržení. V případě velkého množství posluchačů se struktura dramaticky vzdaluje obyčejným strukturám.

PŘEDPOKLAD ZAMÍTNUT

ZÁVĚR

Cílem práce bylo porovnat několik vybraných datových struktur v Jave a JaveFX. Tento proces zahrnoval implementaci pěti datových struktur. Každá struktura byla implementována a následně na ní byly provedeny výkonové testy se základními operacemi jako jsou vkládání, odebírání, přístup dle indexu a přístup dle klíče. Společně s těmi to testy byly provedeny i testy s různými datovými typy, konkrétně s datovými typy Integer a String.

Celkově bylo provedeno asi 66 testů s 5 datovými strukturami, celková doba prováděných výpočtů přesáhla 250 hodin. V tomto čase není zahrnut čas, který byl strávený implementací, přípravou prostředí, vyhodnocením a zpracováním.

Na základě teoretické přípravy bylo vzneseno celkem 6 předpokladů, z toho byly pouze dva zamítnuty, ostatní byly potvrzeny.

Epickým závěrem práce bylo se stalo náhodné vkládání do ArrayListu. Kdy vkládání probíhalo neefektivním způsobem, přerovnáním všech prvků před daným prvkem pomocí cyklu a následný zápis na uvolněné místo. To se ovšem ukázalo jako krajně neefektivní. Proto se tento problém vyřešil optimalizací, kdy se pole nepřerovnává ale rovnou kopíruje do stejně velkého pole, s posunutím indexu, tak vznikne nové místo pro prvek, a ten se v konečném stavu zapíše na místo. Při tomto kopírování bylo využito nativní funkce *System.arrayCopy(...)*. Výsledkem je docílení stejné efektivity jako nativní implementovaná dynamická struktura. Časový rozdíl je opravdu znatelný.

Práce splnila svůj cíl, datové struktury byly korektním způsobem popsány, naimplementovány a otestovány.

POUŽITÁ LITERATURA

AHO, Alfred V, John E HOPCROFT a Jeffrey D ULLMAN. *Data structures and algorithms*. Reading, Mass.: Addison-Wesley, c1983. ISBN 978-0201000238.

EVERITT, Brian. *The Cambridge dictionary of statistics*. 2nd ed. New York: Cambridge University Press, 2002. ISBN 05-218-1099-X.

KEOGH, James Edward a Ken DAVIDSON. *Datové struktury bez předchozích znalostí*. Brno: Computer Press, 2006. ISBN 80-251-0689-6.

KNUTH, Donald Ervin. *Umění programování*. Brno: Computer Press, 2008. ISBN 978-80-251-2025-5.

LEWIS, Harry R a Larry DENENBERG. *Data structures and their algorithms*. New York, NY: HarperCollins Publishers, c1991. ISBN 06-733-9736-X.

MEHTA, Dinesh P a Sartaj SAHNI. *Handbook of data structures and applications*. Second edition. Boca Raton, [2018]. ISBN 978-1498701853.

PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java: kompletní učebnice pro začátečníky*. 2., aktualiz. a rozš. vyd. Praha: Grada, c2009. Knihovna programátora (Grada). ISBN 978-80-247-2653-3.

PROKOP, Jiří. *Algoritmy v jazyku C a C: praktický průvodce*. Praha: Grada, 2009. Průvodce (Grada). ISBN 978-80-247-2751-6.

ROUBALOVÁ, Eliška. *Java bez předchozích znalostí*. Brno: Computer Press, 2015. ISBN 978-80-251-4572-2.

WRÓBLEWSKI, Piotr. *Algoritmy: datové struktury a programovací techniky*. Brno: Computer Press, 2004. ISBN 80-251-0343-9.