



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Aplikace pro správu velké slovní zásoby pod OS Android
Student:	Michal Lepík
Vedoucí:	Ing. Pavel Kubalík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

- 1) Navrhn te a implementujte aplikaci pro OS Android, která bude zam ená na správu slovní zásoby anglického jazyka.
- 2) Aplikace bude umož ůvat zpracovávat velké objemy dat okolo 10 000 sloví ek.
- 3) Zam te se na implementaci algoritmu pro asov efektivní vyhledávání ve slovníku.
- 4) Analyzujte a pro ást využijte Native Development Kit na psaní C kódu pro Android doporu eném pro náro n jší výpo ty a porovnejte výkonnostn s konven ním Software Development Kit p i importu sloví ek z textového souboru.
- 5) Analyzujte a navrhn te vhodný způsob uložení dat na vzdáleném úložišti tak, aby bylo možné efektivn zpracovávat velké slovníky.
- 6) Výkonnost aplikace otestujte na r zných za ízeních a pro r zné objemy dat.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
řídící

V Praze dne 20. ledna 2016

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
KATEDRA TEORETICKÉ INFORMATIKY



Bakalářská práce

Aplikace pro správu velké slovní zásoby pod OS Android

Michal Lepíček

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

16. května 2016

Poděkování

Rád bych tímto poděkoval vedoucímu mé práce, Ing. Pavlu Kubalíkovi, Ph.D., který mou aplikaci průběžně testoval a vznášel konstruktivní návrhy pro směr pokračování práce. Dále bych chtěl poděkovat všem, kteří si stáhli mou aplikaci, otestovali ji a zaslali zpět výsledky výkonnostních testů či obecné připomínky a chyby.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2016

.....

České vysoké učení technické v Praze
Fakulta informačních technologií

© 2016 Michal Lepíček. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Lepíček, Michal. *Aplikace pro správu velké slovní zásoby pod OS Android*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato bakalářská práce se zabývá vytvořením aplikace pro mobilní operační systém Android. Účelem aplikace je správa slovní zásoby cizího jazyka. V práci se zabývám různými metodami vývoje pro Android. Tyto metody jsem analyzoval a podle nich navrhl efektivní postupy pro funkčnost aplikace. Poté těmto postupům měřím čas pro porovnání. Nakonec jsem aplikaci vyvinul do funkčního a stabilního stavu zvládající požadovanou velkou slovní zásobu.

Klíčová slova Android, porovnání algoritmů, slovní zásoba, Java, C/C++

Abstract

This thesis is focused on creating an application for Android, the mobile operating system. The application's purpose is to manage a vocabulary of foreign language. In this work I am dealing with different methods of development for Android. These methods I analyzed and accordingly suggested effective procedures for application functionality. Then I measure time of these procedures for comparison. At last I developed the application into functional and stable state working with required large vocabulary.

Keywords Android, algorithms comparison, vocabulary, Java, C/C++

Obsah

Úvod	1
Cíl práce	2
1 Analýza	3
1.1 Volba platformy	3
1.2 Požadavky	7
1.3 Existující řešení	9
2 Analýza problému	11
2.1 Paměťová náročnost	11
2.2 Výkonnostní testy	11
2.3 Vyhledávání	16
3 Realizace	23
3.1 Formát souboru se slovíčky	23
3.2 Benchmark	23
3.3 Testování slovní zásoby	27
3.4 Databáze a vyhledávání	32
4 Testování	39
4.1 Testování vývojářem	39
4.2 Testování uživateli	40
5 Další rozvoj aplikace	47
5.1 Krátkodobé cíle	47
5.2 Dlouhodobé cíle	47
Závěr	49
Literatura	51

A Seznam použitých zkratek	53
B Obsah přiloženého CD	55

Seznam obrázků

1.1	Architektura operačního systému Android (Zdroj: [3])	4
2.1	Časové porovnání algoritmů v jazyce Java (Zdroj: [17])	22
3.1	Časové porovnání použití SQLite transakcí	24
3.2	Diagram předkládání slovíček při učení	28
3.3	Graf rychlosti mazání z datových struktur	31
3.4	Časové porovnání full-textových modulů SQLite	34
3.5	Časové porovnání vyhledávacích algoritmů	35
3.6	Časové porovnání vyhledávacích algoritmů	36
3.7	Časové porovnání vyhledávacích algoritmů	37
4.1	Znázorněné rozpětí značek testovacích zařízení	41
4.2	Znázorněné rozpětí verzí OS Android na testovacích zařízeních	41
4.3	Časové porovnání Javy a C/C++ dle velikosti dat	42
4.4	Aritmetický průměr výkonnostních testů ve spojnicovém grafu	44
4.5	Aritmetický průměr výkonnostních testů ve sloupcovém grafu	44
4.6	Medián výkonnostních testů ve spojnicovém grafu	45
4.7	Medián výkonnostních testů ve sloupcovém grafu	45

Seznam tabulek

1.1	Zastoupení verzí OS Android k 04. 04. 2016 (Zdroj: [2])	4
2.1	Instrukční sady, pro které je možné zkompileovat C/C++	16
3.1	Rychlost datových struktur používaných jako fronta v ns	30
3.2	Časové porovnání vyhledávacích algoritmů	36
4.1	Časové porovnání vyhledávacích algoritmů v ms	43

Úvod

V dnešní době již není problém navštívit téměř jakoukoliv zemi světa a pro takové turisty je výhodou znát tamní jazyk. Kromě turistických zájezdů je vhodné umět cizí jazyk také kvůli práci v přibývajících mezinárodních společnostech. Není ani tajemstvím, že každý jazyk se vyvíjí a s mizící jazykovou bariérou je vývoj jednoho jazyka ovlivněn druhým. V takových případech se může stát, že starší generace nerozumí výrazům mladší, protože se nevyznají v kořenech daných výrazů.

Dříve se dal učit cizí jazyk z knih nebo od kvalifikovaného učitele, ale každý nemá peníze na zajištění kvalitních lekcí od lektora nebo rodilého mluvčího, ani na neustálé nakupování více a více pokročilejších knih. Nyní, když zažíváme rozmach takzvaných chytrých telefonů, je nezbytné, aby se alespoň část kvalitních lekcí dala učit přímo z těchto mobilních zařízení. Tato mobilní zařízení používají již všechny věkové kategorie včetně dětí a odváznějších seniorů. A právě pro tyto dvě věkové kategorie je třeba se zaměřit na základní správu a výuku cizího jazyka. Nynější děti již vyrostou do světa, který nebude téměř znát jazykovou bariéru a starší občané potřebují cizí jazyk zejména pro jeho prosazování ve všech směrech, například popis tlačítek na dálkovém ovladači od televize nebo elektronický displej na palubní desce auta. Proto obě tyto věkové kategorie potřebují základní správu cizího jazyku ať už se jedná o základ budoucích znalostí nebo nezbytný základ pro fungování v nynějším světě.

Cílem této práce je vytvořit aplikaci, která by zvládala základní správu slovní zásoby cizího jazyka. Přestože je konkurence takovýchto aplikací velká, tak lze jen těžko najít takovou, která by zvládala počet slovíček a frází v takovém množství jaké člověk musí znát, aby si mohl upřímně říci, že zvládá daný jazyk.

Mou motivací je rozšíření povědomí o možnostech vývoje na mobilních zařízeních s operačním systémem Android a v práci se tedy zabývám analýzou, návrhem a implementací aplikace podle různých metod, které následně porovnávám.

Cíl práce

Cílem této práce je vytvoření aplikace fungující na mobilní platformě OS Android. Funkčnost aplikace spočívá ve správě slovní zásoby cizího jazyka a cílem této aplikace je efektivní práce se slovní zásobou čítající větší počet slovíček.

Cílem rešeršní části práce je získání přehledu o programování aplikací pro OS Android a analýza problematiky ohledně zpracovávání většího množství dat na této platformě.

Cílem praktické části práce je návrh efektivních algoritmů použitelných pro správu velké slovní zásoby a jejich následná implementace a vyzkoušení ve funkční aplikaci. Závěrečným bodem praktické části je změření výkonnosti aplikace na různých zařízeních.

Analýza

1.1 Volba platformy

Operační systém Android jsem zvolil, protože je nejrozšířenější mobilní platformou v současné době. Jeho podíl na trhu činí více než 80% [1]. Dalšími faktory ovlivňující tuto volbu je má znalost systému a také, že vlastním zařízení s OS Android, což urychluje vývoj a testování aplikace.

Problémem vývoje aplikace pro OS Android je jeho roztržštěnost po mobilních zařízeních. Kromě různých typů zařízení týkající se hlavně velikosti displeje, paměti a rychlosti procesoru je zde i problém s různou verzí OS Androidu. V tabulce 1.1 můžete vidět zastoupení jednotlivých verzí na trhu.

Od verze OS Android se odvíjí také verze API a je třeba předem zvolit minimální verzi API, kterou by měla aplikace podporovat. Vzhledem k povaze výzkumné práce jsem zvolil minimální verzi API 16 a tedy minimální verze OS Android, na které půjde aplikace nainstalovat, je 4.1.x. Od této verze je v API nejméně změn, které se využívají v běžném kódu a tím odpadnou problémy s častými podmínkami, kdy jedna část kódu se využívá pro starší verze OS Android a druhá část pro novější verze. Navíc je takováto verze teoretickým ukazatelem na stáří a tedy výkonnost zařízení, na kterou se mohou zaměřit.

Existence potenciálních uživatelů se staršími zařízeními je zohledněna v kapitole 5.

1.1.1 Architektura

Informace z této sekce jsou čerpány z [3].

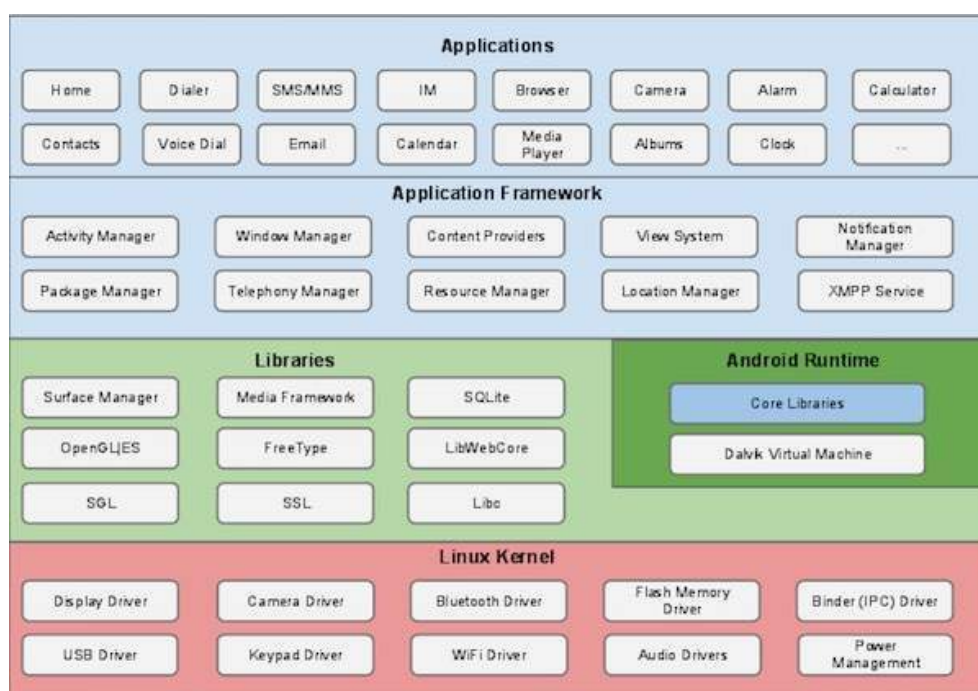
Na obrázku 1.1 je znázorněna architektura OS Android. Architektura je rozdělena do několika vrstev, které jsou rozebrány v následujících podsekcích.

Vyjma následujících existují ještě další, jako jsou abstraktní hardwarová vrstva nebo Binder IPC vrstva. Tyto vrstvy ale nejsou důležité pro vývojáře aplikací. [4]

1. ANALÝZA

Tabulka 1.1: Zastoupení verzí OS Android k 04. 04. 2016 (Zdroj: [2])

Verze	Označení	Podíl
2.2	Froyo	0.1%
2.3.3 - 2.3.7	Gingerbread	2.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	2.2%
4.1.x	Jelly Bean	7.8%
4.2.x		10.5%
4.3		3.0%
4.4	KitKat	33.4%
5.0	Lollipop	16.4%
5.1		19.4%
6.0	Marshmallow	4.6%



Obrázek 1.1: Architektura operačního systému Android (Zdroj: [3])

1.1.1.1 Linuxové jádro

Jádro (Kernel) je základní vrstvou OS Android. Jeho elementární funkcí je zajištění komunikace mezi hardwarem a softwarem. Součástí této vrstvy jsou převážně ovladače (drivers), které zajišťují právě zmíněnou komunikaci. Dále zajišťuje správu paměti, procesů, napájení apod.

Android nepoužívá vlastní jádro, nýbrž upravenou verzi linuxového jádra. První použitá verze byla 2.6.x, aktuální je 3.x a do budoucna se připravuje 4.x dle repozitáře [5] a testovacích zařízení popsaných v sekci 4.1.1.

1.1.1.2 Knihovny

Druhou vrstvou odspoda jsou knihovny, které jsou psané v nativním jazyce C/C++. Tyto knihovny zajišťují základní funkce systému. Například SSL zajišťuje šifrování, SQLite databáze, apod.

1.1.1.3 Android runtime

Android runtime je třetí sekcí architektury a je k dispozici v druhé vrstvě. Obsahuje klíčový prvek Dalvik Virtual Machine (dále jen DVM), což je typ JVM (Java Virtual Machine) používaný pro běh aplikací. JVM je virtuální stroj, který se stará o překlad Javy do nativního kódu.

V roce 2014, kdy byl představen OS Android ve verzi Lollipop (5.0), bylo důležitou novinkou nahrazení DVM novým ART. ART bylo známo již z verze Kitkat (4.4) z roku 2013, ale tehdy šlo pouze o skrytou možnost přepnutí na ART, s kterým mnoho aplikací nepracovalo správně.

DVM byl virtuální stroj, který obsahoval Just-in-Time (dále jen JIT) kompilaci. Start aplikace tehdy doprovázela vždy kompilace a následné spuštění. ART ale přešlo na Ahead-of-Time (dále jen AOT), které kompiluje předem (při instalaci aplikace) a start aplikace tedy obsahuje pouze její spuštění. Výhodou ART je tedy logicky rychlejší start aplikace. Nevýhodou oproti DVM je větší paměťová náročnost.

1.1.1.4 Application framework

Tato vrstva obsahuje další knihovny, tentokrát napsané v jazyce Java. Jde o knihovny, které tvoří systémové API. Skrze toto API může vývojář získat přístup k systémovým prvkům jako jsou tlačítka, k systémovým notifikacím nebo ke sdílenému obsahu jiných aplikací.

Nad touto vrstvou jsou již klasické aplikace, tak jak je známe. Kamera, kontakty, prohlížeč apod.

1.1.2 Vývojové prostředí

Kromě psaní kódu v libovolném textovém editoru a zkompilování skrze příkazovou řádku existují dvě nejznámější vývojová prostředí.

Starší z nich je vývojové prostředí známé jako Eclipse. V roce 2009 byl pro Eclipse vydán plugin ADT, který poskytoval základní funkce, například nový android projekt, vytvoření UI aplikace, ladění a kompilování aplikace.

V roce 2013 bylo společností Google představeno nové vývojové prostředí, které nese název Android studio. Google sám toto prostředí vyvíjí a je založeno na vývojovém prostředí IntelliJ Idea od firmy JetBrains. Android studio je zdarma stáhnutelné ze stránek Android developers a lze ho používat pouze k vývoji aplikací pro OS Android. [6]

Zprovoznění Android studia spočívá ve stažení a spuštění instalačního balíčku. Následně se lze proklikat instalačním dialogem, který za vývojáře stáhne všechny nutné balíčky a knihovny, zejména vývojářské nástroje SDK. Kromě klasických požadavků pro běžný chod prostředí, jako jsou nároky na systém, paměť a rozlišení, je zde také požadavek na JDK. Podporovaná verze JDK se liší dle systému. Zatímco u systémů Windows a Linux se jedná o verzi 7, tak pro Mac je podporována verze 6.

Pro tuto práci bylo zvoleno vývojové prostředí Android studio z důvodu přímé podpory společnosti Google jakožto firmy stojící za samotným OS Android. Dalším faktem je zaměření Android studia na vývoj aplikací právě pro OS Android oproti prostředí Eclipse, které je využíváno pro psaní jakéhokoliv kódu a pro OS Android potřebuje plugin.

Během vývoje byla vydána nová verze Android Studia, která poskočila z verze 1.x na 2.0. Tato verze přináší mnoho změn a hlavní z nich spočívá v novém emulátoru a nové funkci zvané **Instant Run**, která dokáže ukázat změny provedené v kódu bez nutnosti restartovat celou aplikaci. Spolu s novým emulátorem slibuje Google rychlejší vývoj aplikace než s fyzickým zařízením. Kromě toho se také změnila požadovaná verze JDK na nejnovější verzi 8. Nicméně pro Mac zůstala verze 6.

1.1.2.1 Ladění

Spouštět aplikaci během vývoje lze na dvou „typech zařízení“. První možností je emulátor, což je virtualizační program, který simuluje fyzické zařízení v počítači a je jedním ze základních nástrojů dodávaného v SDK. Druhou možností je mít připojené fyzické zařízení prostřednictvím USB. To vyžaduje mít zapnuté **Ladění USB** (anglicky: USB Debugging) ve skrytém nastavení pro vývojáře. Toto skryté nastavení lze odkrýt skrze sedminásobné poklepání na číslo sestavení v nabídce **O telefonu**.

Připojení, přímá instalace se spuštěním a jednoduché ladění k oběma typům zařízení jsou implicitně zprovozněné vlastnosti Android studia.

Obě možnosti jsou obsluhovány službou ADB a na fyzickém zařízení je třeba navíc potvrdit přístup z každého počítače. ADB je také jednou ze součástí SDK.

Emulátor jako virtualizační nástroj byl vždy pomalejší než fyzické zařízení, avšak od nové verze Android studia již tomu tak není, viz poslední odstavec

předchozí sekce 1.1.2.

1.1.3 Databáze a další úložiště

Jeden z typů úložišť využitelných na OS Android je SQLite databáze. Tato databáze je jednou z nativních knihoven v základu systému. SQLite je relační databázový systém, který obsluhuje pouze jediný soubor na jednu databázi.

Soubor s databází se ukládá do interní paměti, kde je přístupný pouze konkrétní aplikaci. Tyto soubory však nejsou nijak šifrované, což může vývojáři pomoci při ladění aplikace.

Verze knihovny SQLite se liší dle verze OS Android, ale někdy výrobci telefonů sami aktualizují tuto knihovnu a tak se nedá spolehnout na žádnou konzistenci. Toto může zapříčinit velký problém pro vývojáře a nezbyvá nic jiného, než za běhu aplikace zjistit verzi SQLite knihovny a podle dostupných možností navrhnout databázi. Nejnovější verzi SQLite, která byla obsažena v OS Android verze N-preview je 3.9.2.

Kromě zmíněné databáze a libovolných souborů v interní či externí paměti, použitelných jako datové úložiště, disponuje OS Android dalším úložištěm pod názvem **Shared preferences**. Jedná se o úložiště na principu „klíč - hodnota“ a může být sdíleno napříč aplikacemi. Využívá se pro jednoduchá data, jako jsou hodnoty položek v nastavení aplikace (wifi - zapnuto/vypnuto).[7]

1.2 Požadavky

1.2.1 Funkční požadavky

1.2.1.1 Lekce

Aplikace umožní uchovávat slovíčka v jednotlivých lekcích. Každá lekce také poskytne základní informace a statistiky.

1.2.1.2 Import a Export

Aplikace umožní import slovíček do aplikace z textového souboru v jednoduchém formátu a export umožní stejnou věc v opačném směru. Oba typy převodu by měly podporovat uchování parametrů, které určují postup uživatele v učení slovíček.

1.2.1.3 Slovník

Aplikace umožní zobrazit seřazený seznam slovíček dle aktuálně vybraných lekcí a vyhledávání v nich.

1.2.1.4 Testování slovní zásoby

Aplikace umožní jednoduché testování znalosti slovíček dle aktuálně zvolených lekcí.

1.2.1.5 Nastavení

Aplikace umožní měnit chování některých svých částí dle volby uživatele. Například: vázanost slovníku s vybranými lekcemi, obtížnost testování, směr překlada.

1.2.1.6 Výkonnostní test

Aplikace umožní uživateli změřit výkon svého zařízení na testech typických pro aplikace na správu slovní zásoby.

1.2.2 Výkonnostní požadavky

Aplikace umožní spravovat velký počet slovíček, který je definován zadáním práce na 10 000. Tento požadavek je mířen na plynulost aplikace při zpracovávání, vyhledávání apod.

1.2.3 Vzdálené úložiště

Součástí zadání je požadavek na vzdálené úložiště pro zpracování velkých slovníků. Nejznámější možnosti jsou služby Google Drive a Dropbox. Obě nabízí vlastní API, které aplikacím dovoluje se přihlásit do těchto služeb.

Tento požadavek se během realizace ukázal jako nepodstatný neboť OS Android nemá problém zvládat větší slovníky a pro komunikaci se vzdálenými úložišti má vlastní vestavěné funkce, které lze využívat.

1.2.4 Ostatní požadavky

1.2.4.1 Android

Aplikace musí běžet v operačním systému Android. Minimální podporovanou verzí byla zvolena 4.1.x známá jako Jelly Bean s verzí API 16. Součástí tohoto požadavku je také dodržení grafických principů podle material design guidelines [8].

1.2.4.2 Offline

Aplikace nesmí být závislá na internetovém připojení.

1.2.4.3 Stabilita

Aplikace by měla být stabilní ve všech možných stavech a neměla by vykazovat chybu, která aplikaci donutí k pádu či jinému nestandardnímu chování.

1.3 Existující řešení

1.3.1 Aplikace

Aplikací na OS Android, které by spravovali uživatelovu slovní zásobu je mnoho. Existují klasické slovníky i pokročilé aplikace na procvičování slov nebo jejich výslovnosti. Ačkoliv je množství takovýchto aplikací velké, tak jsem nenašel aplikaci, která by splňovala všechny mé funkční požadavky plynoucí ze zadání práce. Nejhuře na tom byl požadavek ohledně velkého počtu slovíček. Aplikace ne vždy podporovaly nahrání vlastních slovíček a pokud ano, tak nahrání obrovského počtu bylo příčinou pádu aplikace či jiného selhání. Dalším velkým zklamáním byl počet bezplatných a na internetovém připojení nezávislých aplikací.

1.3.1.1 Bakalářská práce

Zadání mé bakalářské práce vychází z podobného základu jako bakalářská práce Evy Mayerové [9]. Nicméně, její práce byla zaměřena pouze na vytvoření podobné aplikace, kdežto moje zadání míří teoretickým směrem v porovnání metod vývoje a algoritmů na OS Android.

Při vyzkoušení požadavku ohledně 10 000 slovíček ve výsledku zmíněné bakalářské práce jsem narazil na podobný problém. Aplikace sice nepadla ihned, ale spadla kdykoliv jsem chtěl takovýto počet slovíček zpracovávat nebo otestovat. Při menším, ale stále velkém počtu slovíček aplikace sice již nepadala, ale byla velice pomalá.

1.3.2 C/C++ na OS Android

Přes všechnu snahu jsem nenašel literaturu o aplikaci, která by byla na dostatečné úrovni pro studování k mému využití. Vyhledávání slov C++, NDK a podobných mě v obchodu Google play nepřiblížilo k žádné aplikaci, která by opravdu C++ používala. Pravděpodobně vývojáři vůbec tento jazyk pro OS Android nepoužívají nebo to skrývají. Zda-li někdo využívá C/C++ k doporučenému hernímu jádru je též těžko zjištělné a ještě nemyslitelnější je získání takovýchto zdrojových kódů.

Povědomost o používání C/C++ na OS Android je obecně u vývojářů a podobně zaměřené komunity malá, alespoň co se týče mého okolí. Nejčastější reakcí byla nevěřícnost, že je to možné a někteří dotazovaní odpověděli, že o tom někdy někde slyšeli nebo že používali podobný princip v jiné oblasti.

1. ANALÝZA

Na internetu lze nalézt několik návodů na použití C/C++ pro OS Android a pár článků o porovnání jako je [10], avšak úroveň znalostí, které se dají z těchto článků získat, je omezená.

Analýza problému

2.1 Paměťová náročnost

Každá aplikace dostává od systému přidělenou velikost RAM paměti, nazývá se **heap**. Všechny dynamické objekty se ukládají právě na heap, který je jednou za čas pročištěn **Garbage collectorem**. Garbage collector se také spouští, pokud aplikace paměť heap vyčerpá. Vývojář nemůže donutit Garbage collector spustit, může mu jen doporučit, aby začal dělat svou práci; ale v praxi je toto doporučení většinou ignorováno. Pokud jsou nároky na paměť příliš velké a Garbage collector nemůže nic vyčistit, tak aplikaci postihne pád z důvodu chyby **Out of memory**.

Velikost heapu se liší dle verze systému i samotného zařízení a jediný Java způsob pro vývojáře, jak velikost heap paměti ovlivnit, je definovat v aplikaci vlastnost, která bude od systému nárokovat větší než normální velikost heapu. Druhou možností je využít NDK, který tento limit velikosti heapu obchází. Tato práce má za úkol zprovoznit 10 000 slovíček s plynulostí aplikace, což se na novějších zařízeních neukázalo jako problém ani při konvenčních metodách.

Požadavek na 10 000 slovíček a plynulost aplikace lze splnit pouze důvtipnějšími technikami vývoje pro OS Android. Například nepoužívat pro všechno dynamická pole, snažit se vyhnout autoboxingu, nepředávat tato slovíčka skrze parametry, jako tomu bylo u starších verzí OS Android běžným zvykem při používání Aktivit. Důležité je také dát uživateli vědět za pomoci ukazatele průběhu, že něco chvíli potrvá, například získávání většího množství slovíček z databáze.

2.2 Výkonnostní testy

Rozhodl jsem se, že výkonnostní testy (dále jen benchmark) budou měřit nejnáročnější funkci celé aplikace. Touto funkcí je import slovíček. Při importu slovíček se děje několik věcí. Prvně je třeba otevřít soubor, který obsahuje

dvojice slovíček, kterými jsou synonyma v odlišných jazycích (pro tuto práci konkrétně anglický a český jazyk).

Po jednoduché operaci otevřít soubor je třeba dostat z něj tato slovíčka. K tomu slouží dva přístupy. Prvním je projíždět soubor řádek po řádku a za pomoci nejlepších funkcí každý řádek rozdělit na dvě slovíčka. Druhým přístupem je načíst si několik znaků do předem připravené kolekce (známo a dále uváděno jako buffer) a vytvořit automat, který bude po znacích přijímat. Automatem je myšlen lexikální a syntaktický analyzátor. Tento přístup lze dále měnit podle velikosti bufferu. Příliš malý buffer zajistí zbytečně častý přístup na disk a příliš velký donutí aplikaci k vypotřebování heap paměti.

Další operací je vložit slovíčka do úložiště aplikace. Úložištěm pro tuto práci byla zvolena poskytovaná databáze SQLite. Tato operace samotná by nebyla náročná, nebýt smyslem této práce umožnění zpracovávání velkého počtu slovíček. Samotných 10 000 SQL dotazů na vložení řádku do databáze pro každou dvojici slovíček může být náročných. Problém tohoto procesu jsou I/O operace. Když se provádí jeden SQL příkaz (s SQLite) tak se obalí něčím, co je známo jako transakce. Transakce, jednoduše řečeno, umožňuje obalit všechny změny a provést je naráz či je zrušit. Při jedné transakci pro SQLite se děje následující: změna se uloží do paměti, otevře se soubor s databází, aplikují se změny, spouští se funkce `fsync()` pro kontrolu zapsání všech změn a zavření souboru. Při celém tomto procesu SQLite většinu času stráví čekáním na dokončení I/O operací. Vyhnout se tomuto všemu je však jednoduché. Stačí otevřít transakci, provést všechny SQL příkazy a zavřít transakci. Při tomto přístupu se všechny změny uloží do paměti a poté se provedou všechny zápisy najednou, což z 10 000 I/O operací dělá jen jedinou. Teoreticky by se dalo toto zlepšení definovat jako změna z $\Theta(n)$ na $\Theta(1)$. [11]

Celý tento souhrn úkonů je benchmarkem zařízení s mou aplikací. Benchmark bude implementován v konvenčním jazyce Java za pomoci SDK a nekonvenčním C/C++ za pomoci NDK. Více o těchto pojmech v následujících podsekcích.

2.2.1 SDK

Android SDK je souhrn nástrojů používaných pro vývoj aplikací pro OS Android. Pokaždé, když vyjde nová verze OS Android, tak spolu s ní vyjde také nové SDK. Mezi části SDK patří mimo jiné:

- Nezbytné knihovny
- Emulátor
- Ladicí nástroje
- Dokumentace

Aplikace pro OS Android jsou převážně a konvenčně psány pouze v jazyce Java, což umožňuje právě balíček nástrojů SDK. Je tedy samozřejmé, že výkon zařízení by se měl porovnávat také s implementací v tomto jazyce.

2.2.1.1 Externí karty

Během realizace byla zjištěna důležitá součást vývoje aplikací na OS Android s využitím externích pamětí. Benchmark je rozumné spustit a změřit také na tomto typu paměti, ale zdá se, že toto přináší v OS Android obtížnější problém.

OS Android podporoval ze začátku SD karty jako klasické externí paměťové médium. Stejný princip je známý po připojení SD karty nebo i externího pevného disku k počítači. Časem neoficiální vývojáři začali implementovat do OS Android vlastní skripty umožňující aplikace přesouvat na tyto externí karty a stejný princip s jistou omezeností přišel později i do oficiálního OS Android. Problém byl v rychlosti těchto karet. Je všeobecně známo, že interní paměť bývá rychlejší už jen proto, že bývá na základní desce lépe umístěna a dalším podstatným důvodem jsou maximální rychlosti SD karet. Aplikace také nebyly plnohodnotně na kartě, byla zde pouze jejich část, typicky jen dodatečná a vytvářená data. Z těchto důvodů byly aplikace pomalejší, než když byly uloženy na interní paměti a to vedlo časem k odebrání této možnosti z OS Android. Také se přednostně začala vyrábět zařízení bez slotu na tyto karty. V nejnovějším OS Android 6.0 je tomu opět jinak. Tato verze umožňuje označit externí kartu buďto jako vyměnitelné úložiště nebo jako „plnohodnotnou interní paměť“. V prvním případě se karta chová klasicky jako při prvních verzích OS Android, v druhém případě je karta zformátována, šifrována a používána jako součást systému.

Během výše uvedené historie výrobci objevili další možnost, kterou jsem nedokázal přesně vysledovat, ale při realizaci jsem se s ní musel vypořádat. Jedná se o **nevyjímatelnou** externí SD kartu. Sehnal jsem pouze jedno zařízení, které touto možností disponovalo a tak nemohu zaručit stejné pojmenování na všech zařízeních.

Vývojář může funkcemi jednoduše získat cestu ke složce v interním úložišti, která je přístupná pouze dané aplikaci. To je elementární možnost. Další jednoduchou možností je použití funkcí na získání cesty ke složce v externím úložišti (opět výhradní složka pro danou aplikaci). Tímto se dostávám k jádru problému. Tyto funkce vrací cestu v **primárním** externím úložišti, což v případě její existence, je nevyjímatelná externí SD karta. Toto označení ale může být matoucí, jelikož z testů vyplynulo, že se pravděpodobně jedná o stejnou paměť jako interní. A navíc se pravděpodobně jedná pouze o její oddíl. Měření na takovém úložišti nemá tedy žádný význam, pokud již provádím měření přímo v interní paměti.

Problémem je tedy získání cesty k pravému externímu úložišti. Po značném pátrání jsem objevil možnost získat obsah z proměnných prostředí. Funkce

`System.getenv()` si bere za parametr string označující název proměnné, což jsou v mém kontextu názvy `EXTERNAL_STORAGE` a `SECONDARY_STORAGE`. První obsahuje adresu k primárnímu a druhá k sekundárnímu externímu úložišti.

Dalším problémem bylo vytvoření zmíněné složky na správném úložišti. Od jisté verze OS Android nelze tyto složky vytvářet pomocí funkce `mkdir()`. Je nutné použít funkce, které vrací ty správné cesty a pokud neexistují, tak mají práva je vytvořit. Ovšem jak jsem již zmínil, tak jsou vázané k primárnímu úložišti. V novějším API byly přidány stejné funkce, které požadují jako argument cestu. Lze jim tedy dát cestu, kterou jsem zjistil z proměnné prostředí `SECONDARY_STORAGE`. Tyto funkce jsou bohužel omezené pro novější verze OS Android, které nemají ještě dostatečné pokrytí trhu a předem bych se připravil o mnoho testerů a konečně i uživatelů. Naštěstí existují další funkce, které neprováděly přesně to, co bylo třeba a patrně z názvu, ale alespoň vytvořily nezbytné složky. Ověření poté už probíhalo jen na základě pokusu o vytvoření souboru v dané složce a ověření, že se skutečně vytvořil.

Je možné, že existuje verze OS Android nebo atypické sestavení zařízení, které oplývá neemulovanou externí SD kartou, ale je mimo rozpětí funkcí a metod, které jsem popsal výše. V takovém případě je zařízení vyhodnoceno jako bez externího úložiště a aplikace spouští o polovinu méně testů.

2.2.2 NDK

Android NDK je souhrn nástrojů, které umožňují implementovat části aplikace v jazyce jako je C či C++ oproti primárnímu a konvenčnímu jazyku Java. Využití NDK je typické pro herní jádra, simulování fyzikálních jevů a další výpočetně náročné části aplikace. [12]

Android NDK není doporučen pro běžné části aplikace. Prvotním důvodem je razantní zvýšení složitosti aplikace, jelikož nelze napsat celou aplikaci v jiném jazyce než Java. Dalším důvodem je, že použitím jiného jazyka nemusí být dosaženo vyšší rychlosti, protože části aplikace mohou být optimalizované přímo pro systém, čemuž se použitím jiného jazyka vývojář vyhne.

2.2.2.1 Historie NDK

První verze NDK vyšla v červnu roku 2009 a obsahovala GCC kompilátor. Od té doby vyšlo 25 dalších verzí až do května 2015. Během aktualizací bylo přidáno mnoho funkcí, např. podpora OpenGL ES, STL knihovny, C++11, 64-bitová podpora a alternativní kompilátor Clang. [13]

Během vývoje byla vydána nová verze NDK, ve které byl kompilátor Clang povýšen na primární a GCC na alternativní.

2.2.2.2 Instalace a použití NDK

Samotná instalace probíhá stáhnutím komprimovaného balíčku z oficiálních stránek a jeho rozbalením.

Pro jednodušší použití je třeba nastavit systémovou proměnnou `PATH` tak, aby ukazovala do rozbalené složky. Nyní je v terminálu přístupný příkaz **ndk-build**, který po spuštění ve složce s projektem aplikace zkompile kód v C/C++ do patřičných dynamických knihoven (koncovka `.so`) dle zvolených platform a umístí do složky **libs** v hlavní složce aplikace (`/app/src/main`).

Samotný C/C++ kód se píše běžným způsobem a soubory s tímto kódem se ukládají do složky **jni**. Kromě běžných souborů se sem ukládají ještě dva. **Android.mk** a **Application.mk**. První je makefile s vlastní syntaxí. Uvádí se zde hlavně název knihovny a soubory, které se mají zkompileovat. Druhý je nepovinný makefile, který definuje možnost připojit nativní knihovny k C/C++ kódu. Také se zde uvádějí dodatečné přepínače pro kompilátor a takzvané ABI. ABI určuje, pro které instrukční sady se má nativní kód kompilovat. Možnosti jsou uvedené v tabulce 2.1 převzaté z [14]. Čím více instrukčních sad bude aplikace podporovat, tím větší aplikace bude. Během realizace velikost aplikace při zprovoznění NDK (všech instrukčních sad) stoupla z 1,47 MB na 5,44 MB, tedy téměř čtyřnásobek původní velikosti.

Algoritmy, které vývojář napíše v C/C++ kódu jsou napsány ve funkcích. Tyto funkce se pak mohou volat z prostředí konvenční Javy. Aby to bylo možné, je zapotřebí několik kroků. Do makefile se uvádí název knihovny, který je třeba zahrnout ve třídě, kde hodlá vývojář tyto nativní funkce použít. Toto zahrnutí se provádí ve statickém inicializačním bloku za pomoci funkce **System.loadLibrary()**, která má jako argument řetězec nesoucí název zkompileované knihovny. Dalším krokem je deklarovat nativní funkci v Javě. Pokud vývojář chce volat nativní funkci, musí uvést její deklaraci i v dané třídě. Její podoba je stejná jako kterékoliv jiné pouze s přidáním klíčového slova **native**, **public native void nativniFunkce()**. Posledním krokem je umožnění spárování této deklarace funkce a opravdové nativní funkce. To se provádí skrze název funkce v C/C++ kódu. Pokud by předchozí příklad funkce byla reálná funkce, tak by měla název `Java_cesta_k_tride_Trida_nativniFunkce`. prefix `Java_` je vždy napevno, relativní cestu k třídě lze najít na začátku každé takové třídy (tzv. package), třída je název třídy či soubor bez koncovky a následuje název nativní funkce. Vše odděleno podtržítkem (`_`). Funkce, které nejsou volány z Java kódu mohou mít libovolný název. Po provedení těchto kroků a samozřejmě kompilace dokáže aplikace pracovat s C/C++ kódem.

Při realizaci jsem použil přepínač kompilátoru **-std=c++11**, který mi umožňuje psát v mnou preferovaném jazyce C++ verze 11. To skrývá jednoduchý avšak těžko objevitelný problém. C++ na rozdíl od výchozího C pro NDK umožňuje přetěžování funkcí. Jednoduše řečeno: stejný název funkcí s jiným typem či počtem argumentů. Aby toto C++ dokázalo, musí při kompilaci změnit všechny názvy funkcí na jiné, generované a odlišné. Jelikož je třeba dodržet přesný název funkcí, které jsou z Javy volány, tak aplikace spadne s hlášením, že nenašla nativní funkci. Řešením je jednoduché obalení těchto funkcí blokem **extern „C“**.

Tabulka 2.1: Instrukční sady, pro které je možné zkompilevat C/C++

Instrukční sada	ABI hodnota
ARMv7	armeabi-v7a
ARMv8 AArch64	arm64-v8a
IA-32	x86
Intel64	x86_64
MIPS32	mips
MIPS64 (r6)	mips64
Všechny	all

2.2.2.3 NDK a databáze

Vestavěná nativní knihovna SQLite není dosažitelná z kódu v jazyce C/C++. SQLite knihovna je ovšem veřejnosti přístupná jako volné dílo [15] a jako taková se dá zdarma stáhnout a použít, což je jediný způsob jak využít SQLite v NDK. Po stažení knihovny napsané v jazyce C, ji stačí umístit do určené složky a upravit kompilaci tak, aby knihovnu zahrnula do procesu kompilování. Následně je možné využít tuto „vlastní“ knihovnu k přístupu do již vytvořeného souboru s databází dané aplikace (soubor není šifrovaný).

Je nutné si pouze uvědomit, že takovýto postup zapříčiní dvě spojení s danou databází. Jedno spojení z Javy a druhé z C/C++. Je tedy jasné, že například transakce nebudou obsahovat příkazy z obou kódů, ale jen ty, které byly spuštěné ve stejném jazyce. Dalším problémem, který mě napadá, by mohl být kód spuštěný ve více vláknech pracující jak s Javou, tak s C/C++.

2.2.2.4 JNI

JNI je rozhraní, které definuje možnost kódu spuštěným na virtuálním stroji Javy interagovat s nativními knihovnami a programy. [16]

2.3 Vyhledávání

Součástí aplikace je vyhledávání ve slovníku, který obsahuje slovíčka ze všech nebo jen z vybraných lekcí. Tyto lekce a slovíčka je třeba někde uschovávat a nejlepší schéma je podle mého názoru databáze. Databáze, jak už jsem zmínil v sekci 1.1.3, je zajišťována variantou zvanou SQLite.

Slovíčka jsou při zobrazení slovníku zkopírována z databáze a uložena do paměti, aby se zobrazila v seznamu pro uživatele. Tato strategie umožňuje nejen vyhledávání poskytnuté databází ale také využití vyhledávacích algoritmů.

Pro vyhledávací algoritmy je často nutné předzpracování dat. Takové předzpracování může zabrat velkou část paměti, což je pro tuto práci nepřij-

pustné, neboť je třeba zvládnout velké množství slovíček najednou. Předzpracování by toto množství mohlo i zdvojnásobit a tak je třeba dávat pozor, který algoritmus má velkou paměťovou náročnost. Abych nemusel zkoušet implementovat a měřit každý algoritmus, tak jsem se inspiroval u cizího porovnání obshujícího také graf časové náročnosti 2.1 v prostředí Javy [17]. Myslím si, že nejlepšími a nejznámějšími kandidáty jsou Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore a samozřejmě naivní „brute-force“ přístup.

Databáze SQLite nabízí dva typy vyhledávání v textu. Prvním je klíčové slovíčko **LIKE** a druhým je klíčové slovíčko **MATCH**. Oba typy, včetně předchozích algoritmů, jsou podrobněji popsány v následujících podsekcích dle [18].

2.3.1 Brute-force

Naivním algoritmem pro vyhledávání v textu je přístup, kdy porovnávám téměř každý znak s každým. Algoritmus se chová tak, že porovnávám první znak vzorku s textem a posouvám vzorek, dokud se znaky neshodují. Při shodě prvního znaku vzorku a znaku v textu se porovnávají následující znaky v textu i vzorku. Když dojde opět ke shodě, opakuje se předchozí krok s třetími znaky v pořadí. Při neshodě se vzorek posouvá opět o jednu pozici doprava a znovu porovnávám první znaky.

Typické vlastnosti:

- Žádné předzpracování
- Posun vždy jen o jednu pozici
- Časová složitost vyhledávání je $\mathcal{O}(mn)$

Kde m je délka vzorku k vyhledání a n je délka slova k prohledání.

2.3.2 Knuth-Morris-Pratt

Tento algoritmus vyhodnocuje načtené informace efektivněji než naivní přístup. Zatímco v naivním přístupu se při první neshodě porovnání vrací zpět o tolik znaků, kolik se jich shodovalo + 1, tak KMP se vrací jen o tolik, kolik je třeba. Pokud se několik znaků shoduje, tak je přesně známo, které znaky to jsou, jelikož jsou identické s těmi ve vzorku, který hledáme. Je tedy třeba se posunout pouze o několik pozic tak, aby se první znaky v podřetězci a vzorku opět shodovaly. V praxi to znamená, že při vzorku sestaveném z ojedinělých znaků se posune algoritmus vždy o tolik pozic, kolik se shodovalo znaků při posledním pokusu. Počet pozic, o které se algoritmus posouvá, se dá spočítat předem.

Typické vlastnosti:

- Předzpracování se provádí v lineárním čase $\mathcal{O}(m)$
- Předzpracování má lineární paměťovou náročnost $\mathcal{O}(m)$
- Časová složitost vyhledávání je $\mathcal{O}(n)$

Kde m je délka vzorku k vyhledání a n je délka slova k prohledání.

2.3.3 Rabin-Karp

Tento algoritmus je založen na porovnávání hashe místo jednotlivých znaků. Předzpracování sestává z vytvoření hashe podle vzorku textu, který má být vyhledán. Následně při vyhledávání se vytváří hashe z textu, ve kterém se vyhledává a porovnává se jejich podobnost. Hash každého podřetězce je definován vztahem $\text{hash}(w[0 \dots m-1]) = (w[0] * 2^{m-1} + w[1] * 2^{m-2} + \dots + w[m-1] * 2^0) \bmod q$, kde w je slovo o délce m a q je velké číslo, typicky prvočíslo.

Typické vlastnosti:

- Používá hashovací funkci
- Předzpracování se provádí v lineárním čase $\mathcal{O}(m)$
- Předzpracování má konstantní paměťovou náročnost
- Nejhorší časová složitost vyhledávání je $\mathcal{O}(mn)$
- Průměrná časová složitost vyhledávání je $\mathcal{O}(m + n)$

Kde m je délka vzorku k vyhledání a n je délka slova k prohledání.

2.3.4 Boyer-Moore

Tento algoritmus porovnává opět jen znaky vzorku a textu. Rozdíl oproti KMP je, že některé znaky úplně přeskočí. U KMP se každý znak zpracovává alespoň jednou, u naivního algoritmu dokonce mnohokrát. BM porovnává vzorek oproti textu zprava doleva a k naivnímu posunu oplývá ještě speciálními dvěma posuny. **Good-suffix** posun a **bad-character** posun.

Při neshodě znaků vrací bad-character číslo udávající počet pozic, o které se má vzorek posunout doprava. Pokud vzorek obsahuje znak, který se shoduje se znakem v textu, na kterém byla aktuálně nalezena neshoda, tak bad-character vrací takové číslo, aby se tyto dva znaky zarovnaly. Pokud vzorek takový znak neobsahuje, tak bad-character vrací takové číslo, aby se první znak vzorku zarovnal s následujícím znakem textu po neshodném.

Good-suffix vrací také počet míst, o kolik se má vzorek posunout doprava. Při neshodě znaků vrací takové číslo, aby se vzorek posunul tak, aby dosud

shodné znaky byly opět shodné, ale zároveň s jiným předcházejícím znakem. Například, shodují se dva znaky yz , kterým v textu předchází znak w , ale ve vzorku jim předchází x . Nyní good-suffix doporučí posun vzorku tak, aby znaky yz v textu byly stále shodnými se znaky ve vzorku, ale zároveň jim nepředcházel znak x . Pokud takový segment znaků ve vzorku už není nastává zmenšování segmentu. Nalezne se nejdelší podsegment v textu (dle příkladu, ze segmentu obsahující znaky yz) zprava tak, aby se shodoval se segmentem ve vzorku zleva. Neboli dle příkladu je možné nalézt pouze podsegment obsahující znak z ve vzorku a ten zarovnat nebo takový podsegment nenalézt a posunout vzorek za segment.

Při naivním postupu tohoto algoritmu a nalezení neshody se algoritmus zeptá good-suffix a bad-character posunu na jejich doporučení. Jelikož bad-character může vrátit záporné číslo, což by znamenalo nesmyslný posun vzorku doleva, tak si algoritmus vybírá z těchto dvou čísel maximum a tím dochází k přeskokování znaků zmíněném v úvodu tohoto algoritmu.

Typické vlastnosti:

- Porovnává zprava doleva
- Předzpracování se provádí s časovou složitostí $\mathcal{O}(m + k)$
- Předzpracování má paměťovou náročnost $\mathcal{O}(m + k)$
- Časová složitost vyhledávání je $\mathcal{O}(mn)$
- Nejlepší časová složitost je $\mathcal{O}(n/m)$

Kde m je délka vzorku k vyhledání, n je délka slova k prohledání a k je konečná délka abecedy.

2.3.5 SQLite LIKE

První a základní možností, kterou poskytuje databáze pro vyhledávání, je klíčové slovíčko LIKE uváděné v podmínce dotazu. Nejjednodušší dotaz má podobu `SELECT * FROM tabulka WHERE z LIKE x`. Zastupující znak x v uvedeném příkladě představuje text, který ve sloupci z chceme najít. Při použití LIKE se provádí porovnávání každého řádku databáze a tudíž nemusí být takové vyhledávání nikterak rychlé.

Pokud se zadá pouze jednoduchý řetězec k vyhledání, tak LIKE vyhledá řádek, ve kterém daný sloupec obsahuje pouze tento řetězec. Pro hledání daného textu jako podřetězce existují takzvané „wildcards“. Existují dvě. První je znak procenta (%), který může zastupovat libovolný počet jakýchkoliv znaků včetně nulového počtu. Druhá je znak podtržítka (_), který zastupuje přesně

jeden znak. „LIKE Micha_%“ tedy hledá řetězce začínající na *Micha* a obsahující ještě minimálně jeden znak, například *Michal*, *Michaela*, *Michael úpěl ďábelské ódy*.

LIKE může být zkombinován s klíčovým slovíčkem **NOT**, které se píše před něj a zapříčiní negaci podmínky.

Tato možnost vyhledávání je elementární funkcí každého databázového systému a lze ji tedy použít s jakoukoliv verzí SQLite instalovanou v OS Android.

2.3.6 SQLite MATCH

Informace v této sekci jsou čerpány z [19].

Druhou databázovou možností na vyhledávání v textu je klíčové slovíčko **MATCH**. Tato možnost je založena na full-textovém vyhledávání, což je speciální způsob vyhledávání v databázích či podobných strukturách v nejkratším možném čase.

Rozdíl oproti možnosti LIKE spočívá v rychlosti a způsobu vyhledávání. **SELECT * FROM tabulka WHERE tabulka MATCH 'x'** je nejjednodušší dotaz. Lze si všimnout, že zde není podmínka uvalená na jeden sloupeček, nýbrž na celou tabulku.

Řetězec *x* může mít opět několik podob. Základem jsou zastupující znaky hvězdička (*) a stříška (^). Pokud napíšu jednoduchý řetězec *linux*, tak **MATCH** vyhledá všechny řetězce obsahující slovo *linux*. Pokud má řetězec podobu *lin**, tak **MATCH** vyhledá všechny řetězce obsahující slova *linux*, *lingvistika* apod. Stříška označuje začátek řetězce. Tedy řetězec v podobě *^linux* znamená nalezení všech textů, kde prvním slovem je *linux*. Řetězec v kombinaci *^lin*apl** vyhledá všechny řetězce, kde text začíná slovem začínající na *lin* a druhým slovem začínající na *apl*, například „linuxové aplikace“ nebo „lineární aplikátory“.

Dalším klíčovým slovíčkem, které se zapisuje přímo do vyhledávacího řetězce, je **NEAR**. Toto slovíčko se píše mezi dvě slova (možno včetně zastupujících znaků) a označuje vyhledávání textů, kde se tato dvě slova nachází blízko sebe.

Další možností, jak zapsat vyhledávací řetězec, je specifikování sloupce. Sloupce se zapisují též přímo do vyhledávacího řetězce. Příkladem je vyhledávací řetězec v podobě *titulek:linux ^aplikace*. Tento řetězec vyhledá všechny řádky, kde alespoň jeden sloupec má první slovo *aplikace* a zároveň sloupec titulek obsahuje slovo *linux*.

Aby full-textové hledání fungovalo, je třeba v tabulkách daným sloupečkům přidat speciální index. V SQLite se tento index vytváří spolu se speciální tabulkou *ftsX*, kde *X* nabývá hodnot 1 až 5 dle verze, kterou chce vývojář použít. Syntaxe vytváření tabulky s tímto full-textovým indexem je takováto: **CREATE VIRTUAL TABLE nazev USING FTSX(sloupec)**.

Tyto *fts* tabulky mohou být napojené na obyčejné tabulky, které se používají na vše, kromě vyhledávání. Pokud vývojář zvolí takový postup, tak musí

udržovat obě tabulky konzistentní. Trigger většinou lze v takových případech použít.

2.3.6.1 FTS1 a FTS2

Tyto dvě verze jsou zastaralé, obsahují chyby a doporučuje se je nepoužívat.

2.3.6.2 FTS3 a FTS4

Tyto dvě verze jsou aktuálně používané a jsou téměř identické. Verze 4 má samozřejmě vylepšené části kódu. Měla by být rychlejší a například zastupující znak stříška (^) funguje až od této verze.

Důležitým faktorem při výběru mezi těmito dvěma verzemi je, zda vůbec OS Android obsahuje verzi SQLite s FTS4 modulem. FTS3 je k dispozici od SQLite verze 3.5.0, což zhruba odpovídá první veřejné verzi OS Android - Cupcake a tedy je FTS3 dostupný na všech zařízeních. Kdežto FTS4 byl přidán do SQLite verze 3.7.4, která byla obsažena až v OS Android verze Ice Cream Sandwich. Navíc, jelikož mnoho neoficiálních portů OS Android obsahuje zastaralé verze SQLite, tak je pro vývojáře stále nutné zjistit aktuální verzi SQLite za běhu aplikace a podle toho se řídit.

Nejdůležitějším rozdílem je možnost použít externí obsah od verze 4. Část příkazu „...tab2 USING FTS4(content=tab1, titulek)“ znamená, že virtuální tabulka tab2 má odkaz na obyčejnou tabulku tab1 a obsahuje pouze sloupec, na kterém se má vytvořit index, v tomto případě titulek. Zároveň není třeba při upravování originální tabulky (upravovat řádky, přidávat řádky, mazat řádky, apod) upravovat i virtuální tabulku, protože jsou propojené. Jediné co je třeba pro zajištění konzistence, je po úpravě (případně před vyhledáváním) spustit nad virtuální tabulkou příkaz `INSERT INTO tab2(tab2) VALUES('rebuild')`, který kompletně smaže a vytvoří znova celý index. Jelikož tento rozdíl ušetří jistou duplikaci dat a dle [19] je FTS4 srovnatelně rychlá s FTS3, je pro tuto práci FTS4 první verzí, kterou je třeba vyzkoušet a změřit.

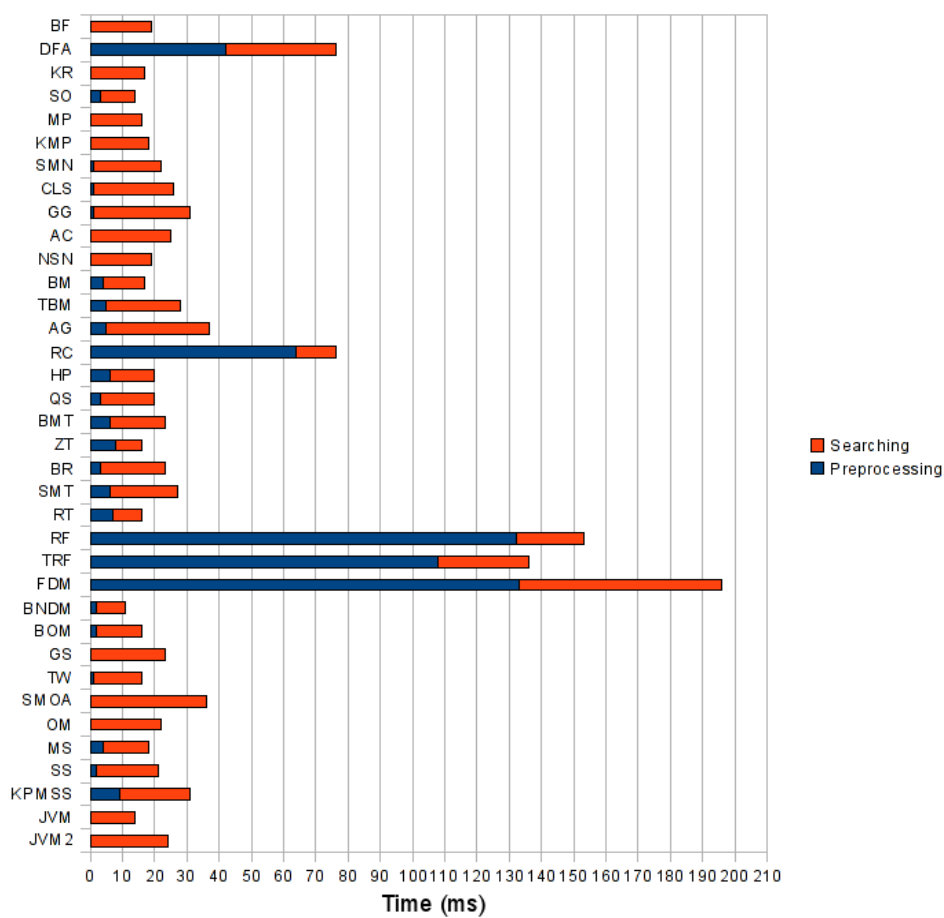
2.3.6.3 FTS5

FTS5 je poslední veřejná verze, která je obsažena v SQLite verze 3.9.0. Tuto verzi má pouze nejnovější OS Android verze Marshmallow, testovací verze N a pravděpodobně také některé upravované starší verze.

Kromě klasického vylepšování je v této verzi rozdíl pro velká data. Verze 3 a 4 nezvládali seznam dat, jehož délka přesáhla limit uvedený v konstantě `SQLITE_MAX_LENGTH`. Verze 5 takto dlouhé seznamy dat rozděluje na menší.

Zároveň byla verze 5 zbavena několika funkcionalit, na druhé straně bylo přidáno API pro vývojáře, díky kterému si mohou dopsat jakékoliv vlastní funkcionality.

2. ANALÝZA PROBLÉMU



Obrázek 2.1: Časové porovnání algoritmů v jazyce Java (Zdroj: [17])

Z pohledu této práce není důvod nevyzkoušet časovou výkonnost této verze.

Realizace

3.1 Formát souboru se slovíčky

Součástí zadání práce je podpora importu a exportu slovíček z/do souboru v jednoduchém formátu. Požadavek jednoduchého formátu jsem splnil umístěním každé dvojice slovíček (český a anglický překlad) na jeden řádek a samotné oddělení dvojice za pomoci znaku rovná se (=). Prvním slovíčkem je anglický překlad a druhým je český překlad.

Ukázka formátu souboru:

```
hello=ahoj  
dog=pes  
...
```

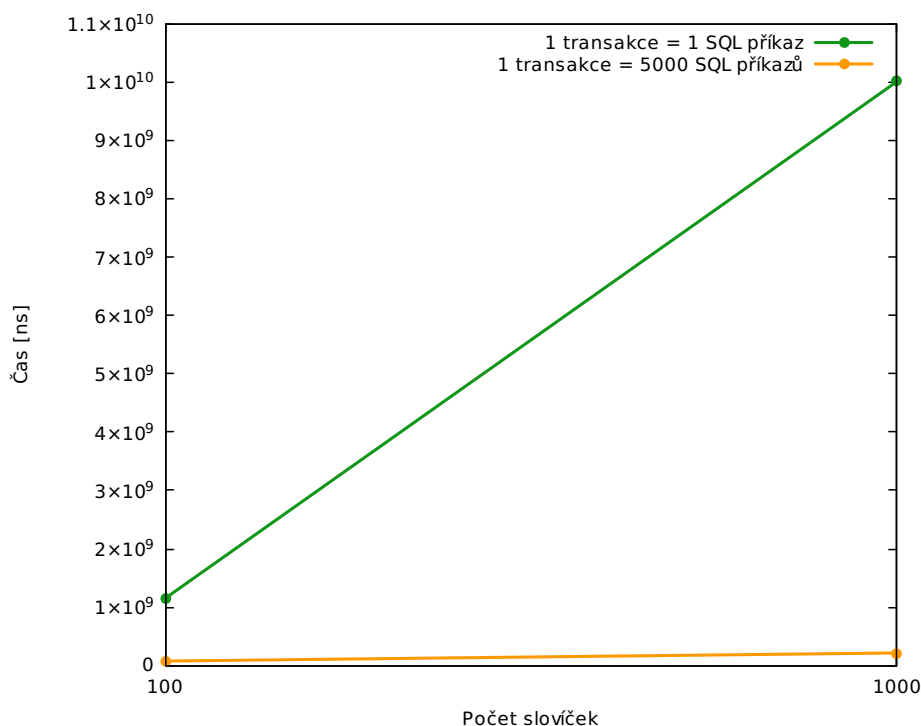
Tento jednoduchý formát umožní snadnou orientaci i vytvoření vlastního souboru jakkoliv zkušenému uživateli. Nevýhodou je nemožnost využít znak rovná se ve slovíčku samotném, o čemž se zmiňuji v kapitole 5.

3.2 Benchmark

V sekci 2.2 jsem nastínil provedení benchmarku. Konkrétně se jedná o změření času, který aplikace stráví nad importem slovíček z textového souboru.

3.2.1 Import

Import slovíček zahrnuje operace uložení nové lekce do databáze, otevření souboru, načtení a zpracování slovíček, jejich uložení do databáze a zavření souboru. Samotné ukládání slovíček do databáze je prováděno přes transakce. Jedna transakce obaluje 5 000 SQL příkazů na vložení, aby nedošla heap paměť.



Obrázek 3.1: Časové porovnání použití SQLite transakcí

Rozdíl mezi rychlostmi při použití a nepoužití transakce je už při malém počtu slovíček tak znatelný, že jsem změřil tento rozdíl pouze párkrát interně a do benchmarku jej vůbec nezahrnoval. Rozdíl lze vidět v grafu 3.1 na 100 a 1 000 slovíčkách. Je patrné, že už u sta slovíček je doba importování 1 sekunda bez transakce a zlomek sekundy s transakcí. U tisíce slovíček je rozdíl mnohonásobně markantnější. Proč tomu tak je, jsem vysvětlil v sekci 2.2.

3.2.1.1 Řádek po řádku

Prvním typem importu je parsování řádek po řádku. V tomto typu se provádí pouze otevření souboru a čtení řádek po řádku. Jeden řádek se následně rozdělí podle znaku rovná se (=) a z toho vzniknou dvě slovíčka. Nejelegantnější způsob, který Java pro rozdělení řetězce nabízí, jsou třídy **Matcher** a **Pattern**. Pro pattern stačí zkompilovat regulární výraz, který označuje části, které chceme v řetězci najít. V mém případě se jedná o regulární výraz „(.+)=(.+)“. Pattern poté při získání řetězce vrátí Matcher, který obsahuje funkci **group()**. Tato funkce má jako argument číslo, které označuje požadovanou část řetězce. Toto číslo se odvíjí od pořadí a vnoření párů závorek. V mém případě jde o jednoduchá čísla 1 a 2. Jednotlivá slovíčka jsou pak zpracována do objektu a uložena do databáze.

Domnívám se, že nejjednodušším ekvivalentem pro výše zmíněné funkce v jazyce C/C++ je objekt **stringstream** a funkce **getline()**. Funkce **getline()** má jako první argument objekt **stringstream**, který v sobě má řetězec. Pokud se specifikuje nepovinný druhý argument, kterým je znak, tak **getline()** vrátí řetězec pouze do tohoto znaku exkluzivně. Pokud se nspecifikuje, tak se čte do znaku nového řádku. Výsledkem aplikování zmíněné funkce a objektu na můj případ je posloupnost příkazů: vytvoření objektu **stringstream** s jednou řádkou jako řetězec, zavolání funkce **getline()** s objektem **stringstream** a znakem rovná se (=) a znova zavolání stejné funkce bez druhého argumentu.

3.2.1.2 Automat

Druhým typem importu je automat. Pojmem automat označuji dvojici lexikálního a syntaktického analyzátoru. Menší součástí je také má vlastní třída zajišťující čtení ze souboru.

Čtecí třída zajišťuje největší rozdíl mezi tímto a prvním typem importu. Tato třída obsahuje tzv. **buffer**, do kterého se načítají znaky ze souboru bez omezení na konce řádků. **Buffer** je omezený pouze svou velikostí, která je nastavená na výchozích 4096 znaků. Smyslem **bufferu** je snížit počet přístupů na disk.

Na začátku procesu se nahraje do **bufferu** 4096 znaků, které si následně po jednom odebírá lexikální analyzátor. Pokud jsou znaky vyčerpány, přistoupí se na disk do souboru a načte se dalších 4096 znaků až do jejich vyčerpání. Lexikální analyzátor tyto znaky shromažďuje a vytváří z nich tokeny. Tokeny, které jsou potřeba pro můj import, jsou pouze slovíčko, znak rovná se, znak nového řádku a konec souboru.

Tyto tokeny se vytváří na žádost syntaktického analyzátoru, takže v celém procesu je vždy alokován pouze jeden token, který je právě zpracováván. Tento analyzátor zjišťuje, zda má soubor správný formát. Provádí to následujícím způsobem: má token „slovíčko“, musí přijít token „rovná se“. Pokud nepřijde, tak obecně skončí chybou. V mém případě přeskočí celý řádek a pokusí se načíst zbytek souboru. V tomto analyzátoru se již řeší také vložení do databáze. Pokud přijdou tokeny „slovíčko“, „rovná se“, „slovíčko“ v tomto pořadí, tak má analyzátor dvojici, kterou může uložit.

Během realizace syntaktického analyzátoru jsem narazil na omezení z hlediska **heap** paměti. Prvním typem implementace syntaktického analyzátoru je rekurzivní sestup, který je v tomto případě lépe pochopitelný pro lidský mozek. Během testování se ale ukázalo, že počet uložených referencí na volající funkce je ohromný a **heap** paměť dojde po pár zpracovaných slovíčkách. Řešením byl přepis na iterativní algoritmus, který nezaplňuje **heap** paměť.

3.2.2 Měření

OS Android poskytuje dvě možnosti měření času části kódu. Jde o funkce zvané `nanoTime()` a `currentTimeMillis()`. Obě spadají do třídy `System` a vrací číslo v datovém typu `long` udávajícím aktuální čas v nanosekundách. `NanoTime()` vrací aktuální čas dle nastaveného pevného bodu, kdežto `currentTimeMillis()` je závislý na reálném čase a tedy náchylný například na synchronizaci času podle internetu nebo na přechodu na letní čas. Pro svou práci jsem zvolil `nanoTime()`.

K měření jsem využil pomocnou třídu `AsyncTask`, která obsahuje tři důležité metody. První je metoda `onPreExecute()`, která se spouští v hlavním vlákne předem. Tu využívám pro zaznamenání startovního času. Druhou metodou je `doInBackground()`, která se spouští na pozadí ve vlastním vlákne a nese v sobě kód, který chci měřit. Třetí metodou je `onPostExecute()`, která se volá po dokončení druhé metody a běží již opět v hlavním vlákne. Tuto metodu používám kromě jiného pro získání aktuálního času, od kterého odečtu první získaný čas. Takto získám dobu běhu, kterou využívám v benchmarku.

Kromě těchto hlavních metod existuje ještě jedna užitečná, známá pod názvem `onProgressUpdate()`, která běží na hlavním vlákne a vývojář je schopen ji zavolat z vedlejšího vlákna metody `doInBackground()`. Využívám ji pro aktualizaci ukazatele průběhu, aby uživatel viděl, jak rychle aktuální test probíhá. Rozdělení na vedlejší a hlavní vlákna má kromě klasických důvodů na OS Android ještě jeden speciální. Hlavní vlákno je takzvané UI vlákno, které se stará o prvky na obrazovce a vedlejší vlákna k těmto prvkům nemají většinou přístup.

Výše jsem zmínil operace, které se provádějí při benchmarku. Jedná se o `import`, který je obohacen ještě operací smazání slovíček z databáze, jelikož se jedná pouze o testovací slovíčka. Všechny tyto operace obaluje jeden test a těchto testů je dohromady 6. Prvním je `import` řádek po řádku, dalších pět jsou automaty. Těchto pět automatů se liší velikostí bufferu, které jsou následující: 32, 256, 2048, 16384 a 65536. Velikosti byly určeny náhodně tak, aby je zvládla jakkoliv velká heap paměť, pokrývaly větší rozsah a byl mezi nimi dostatečný rozdíl. Vyzkoušel jsem i další s vyšším rozdílem a extrémnější velikostí, ale neprokázaly se jako dostatečně odlišné, aby je bylo třeba měřit. Těchto šest testů jsou základním kamenem, který se zdvojnásobuje pro měření v jazyce C/C++. Každý z těchto testů je měřen minimálně dvakrát, jednou v jazyce Java a jednou v ekvivalentní implementaci v jazyce C/C++. Pokud testované zařízení vlastní obsazený slot na externí SD kartu, tak se počet testů znovu zdvojnásobí na celkových 24 testů, při kterých se každý test provede 4x. Jednou v Javě na interní paměti, podruhé v Javě na externí paměti, potřetí v C/C++ na interní paměti a počtvrté v C/C++ na externí paměti.

Rozdíl mezi implementací v jazyce Java a C/C++ jsem se snažil srazit na minimální a dle vlastního uvážení se mi to podařilo. Při samotném měření existuje pouze rozdíl v aktualizaci ukazatele průběhu. Zatímco v konvenční

Javě existuje výše uvedená metoda `onProgressUpdate()`, kterou lze volat při každé zpracované dvojici slovíček, tak v C/C++ takto jednoduchá metoda není. Pro aktualizování průběhu při testu v jazyce C/C++ jsem zvolil nativní funkci, která vrací počet již zpracovaných slovíček a kterou volám opakovaně z vedlejšího vlákna Javy každých 50 milisekund. Číslo, které nativní funkce vrátí pak předávám již klasicky metodě `onProgressUpdate()`.

Všechny testy probíhají s jediným souborem obsahující 50 000 dvojic slovíček. Dle zadání této práce bych měl otestovat zařízení na různé objemy dat, ale při realizaci se ukázalo, že naměřený čas jednoho importu je přímo úměrný k počtu slovíček a tak nemá cenu přidávat tuto variaci do benchmarku, která by jej jen prodloužila s předem známým výsledkem. Menší počet testů má také účel pro testovací komunitu, která by kvůli delšímu času stráveným nad benchmarkem mohla ztratit zájem o jeho dokončení.

Po dokončení benchmarku se uživateli ukáže jednoduchý graf popisující časové průběhy jednotlivých testů a dvě možné akce. První je zopakovat benchmark a druhou je odeslat data vývojáři, mně. Pro přenos jsem zvolil jednoduché odeslání přes internet. Naměřená data jsou zabalená v objektu JSON a přijímána jednoduchým PHP skriptem, který jsem napsal. Kromě naměřených dat se odesílají také anonymní identifikační údaje zařízení (model telefonu, verze OS Android a řetězec známý jako fingerprint popisující verzi sestavení systému) a hash, který mi dovolí kontrolovat validitu odesílaných dat na straně přijímacího serveru.

3.3 Testování slovní zásoby

V části pro testování slovní zásoby jsou dvě metody na procvičování. Obě mají vlastní přístup k průběhu předkládání slovíček uživateli.

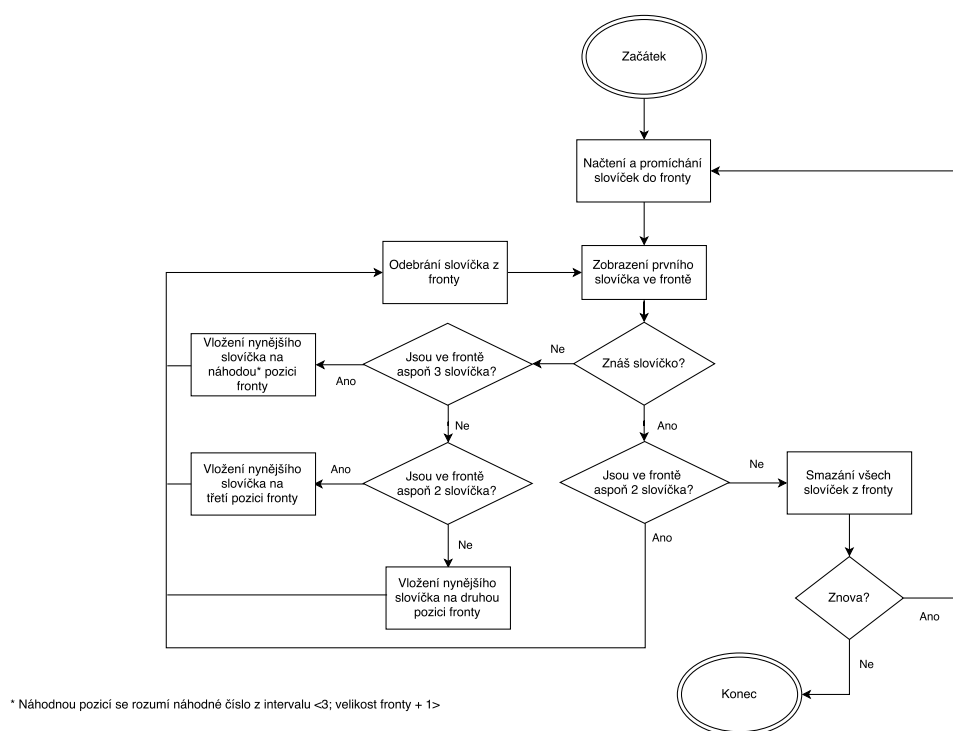
3.3.1 Učení

Učení je typ procvičování, kdy se uživateli zobrazí slovíčko a on sám si rozhodne zda slovo zná či ne. Uživatel má k dispozici tři interakční tlačítka. První zobrazuje překlad pro uživatelovo ujištění a další dvě slouží pro určení znalosti respektive neznalosti slovíčka. Při neznalosti se dané slovíčko objevuje v náhodném pořadí znovu dokud se uživatel slovíčko nenaučí.

Metoda učení je plně závislá na aktuálně zvolených lekcích a lze pouze změnit směr překladu.

Slovíčka jsou objekty uložené v kolekci zvané `LinkedList` (obousměrný spojový seznam), která disponuje možností vkládat prvek na jakékoliv místo v nejhůře lineárním čase $\mathcal{O}(n)$. Přestože se zobrazuje vždy první slovíčko z kolekce a stačila by tedy fronta, tak je třeba znovu předložit v budoucnu slovíčko, které uživatel označil jako neznámé. `LinkedList` je třída implementující abstraktní datový typ `List`, který je nezbytný pro metodu Javy na náhodné promíchání objektů v kolekci. Tato metoda zvaná `shuffle()` je ze třídy `Collections` a

3. REALIZACE



Obrázek 3.2: Diagram předkládání slovíček při učení

má časovou složitost $\mathcal{O}(n)$, kde n je počet prvků v Listu. Algoritmus tohoto procvičování včetně zobrazování slovíček lze vyčíst na diagramu 3.2 a je z něj zřejmé, že každá otázka má nejhorší časovou složitost $\mathcal{O}(n)$.

Paměťová složitost je $\mathcal{O}(n)$, kde n je počet slovíček z aktuálně vybraných lekcí.

3.3.2 Zkoušení

Zkoušení je typ procvičování, kdy uživatel volí překlad zobrazeného slovíčka z nabízených možností. Toto procvičování je závazné pro různé statistické údaje vázané k lekcím. Proto je zkoušení obdobou klasického testu známého ze školního prostředí - jeden pokus na otázku, jedna z možností je správná. Při špatné volbě překladu se dané slovíčko v testu již nezobrazí a o tom, zda je odpověď správná se lze dozvědět z barevného indikátoru umístěném ve spodní části obrazovky.

V metodě zkoušení lze kromě směru překladu změnit i počet možností překladu. Počet možností lze změnit v nastavení na 3, 4 nebo 5. Kromě počtu lze možnosti ovlivnit i nastavením zvaným „Obtížné testování“. V zapnutém stavu se možnosti negenerují pouze z aktuálně vybraných lekcí, ale ze všech dostupných.

Metoda zkoušení je oproti učení jednodušší, avšak vyžaduje větší paměťovou náročnost z důvodu zaznamenání výsledků. Časová složitost zkoušení je pro každou otázku $\mathcal{O}(\log l)$ a celková paměťová složitost je $\mathcal{O}(k + l + m + n)$, kde $3 \leq k \leq 5$ dle volby uživatele, l je počet aktuálně zvolených lekcí, m je počet slovíček pro generování možností a n je počet slovíček na odzkoušení. Detaily jsou popsány v následujících podsekcích.

3.3.2.1 Možnosti

Všechna slovíčka, která jsou nabízena jako možnosti překladu, jsou uložena v dynamickém poli, kvůli konstantnímu $\mathcal{O}(1)$ přístupu k jakémukoliv prvku. To je nezbytné pro generování 3 - 5 možností na každou otázku. Celé toto pole má paměťovou složitost $\mathcal{O}(m)$, kde m je počet slovíček pro možnosti.

Generování náhodných možností z celého pole na jednu otázku bylo nejdříve implementováno promícháním a následně vybráním nutného počtu slovíček ze začátku pole. Takový postup měl časovou složitost $\mathcal{O}(m)$, kvůli prvotnímu promíchávání při každé nové otázce. Nicméně následné smazání promíchávání pole a vybírání náhodných možností pomocí generátoru náhodného celého čísla coby indexu, se časová složitost srazila na $\mathcal{O}(1)$.

Typické ukládání momentálních možností pro danou otázku do ArrayListu s prvky typu celého objektu slovíčka bylo nahrazeno jednoduchým statickým polem řetězců. Paměťová složitost tak sice zůstala na $\mathcal{O}(k)$, kde $3 \leq k \leq 5$, ale byla tím zrušena nadbytečná režie i nadbytečná data.

3.3.2.2 Slovíčka

Slovíčka, která budou postupně předkládána jako otázka pro překlad jsou po promíchání uložena do fronty. Pro tuto frontu je využita datová struktura ArrayDeque, která implementuje abstraktní datový typ Queue (fronta). Při zkoušení se špatně zodpovězená slovíčka znovu nezobrazují a tak není třeba řešit časovou složitost přidání do pole na náhodné místo a je vhodné zvolit frontu. Kromě mazání ze začátku fronty je potřeba také častý přístup k prvnímu prvku fronty. Obojí má konstantní časovou složitost $\mathcal{O}(1)$.

Vzhledem k používání pouze těchto operací by dávalo smysl použít LinkedList (obousměrný spojový seznam) namísto ArrayDeque, který používá pole pro uložení prvků. Nicméně, manuál ArrayDeque zmiňuje „*This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.*“ [20], což potvrzuje i zkouška zaznamenaná v tabulce 3.1. Problém vidím v režii při vytváření nového objektu pro spojový seznam a také by velkou roli mohl hrát **Garbage collector**, který v závislosti na implementaci projíždí při smazání prvku spojového seznamu i všechny ostatní napojené. Při typickém použití ArrayListu jako fronty se časová náročnost mazání prvků mnohonásobně zvýší, protože je třeba posouvat všechny prvky doleva, což odpovídá časové složitosti $\mathcal{O}(n)$, ale pokud se maže odzadu, tak

Tabulka 3.1: Rychlost datových struktur používaných jako fronta v ns

Datová struktura	Inicializace	Vložení prvků	Smazání prvků
ArrayList L	14 739	372 656	22 560 885
ArrayList LC	42 708	167 865	26 208 074
ArrayList LB	17 761	119 583	153 697
ArrayList LCB	29 531	107 708	154 635
LinkedList Q	61 302	2 038 437	121 354
ArrayDeque Q	20 260	1 070 991	92 656
ArrayDeque QC	40 729	1 085 104	96 354

L - Implementuje List

C - Předem stanovená kapacita

Q - Implementuje Queue

B - Operace prováděné z konce

není třeba nic posouvat a časová složitost je konstantní $\mathcal{O}(1)$. Každopádně je stále pomalejší než ArrayDeque.

Ve zkoušení je mazání ze začátku respektive konce nejnáročnější operací a proto jsem do grafu 3.3 zaznamenal její rychlost se zvyšujícím se počtem slovíček. V grafu lze vidět, že inicializace počáteční velikosti nemá až tak velký vliv. ArrayList s mazáním zepředu se složitostí $\mathcal{O}(n)$ se očekávaně nemůže od 1 000 prvků měřit s ostatními datovými strukturami. Naopak LinkedList se drží těsně pod ArrayListem s mazáním odzadu a až u milionu prvků se oproti němu zpomalí. Z praktického hlediska se ale téměř všechny hodnoty drží pod jednou vteřinou, což je pro uživatele dostatečně rychlé.

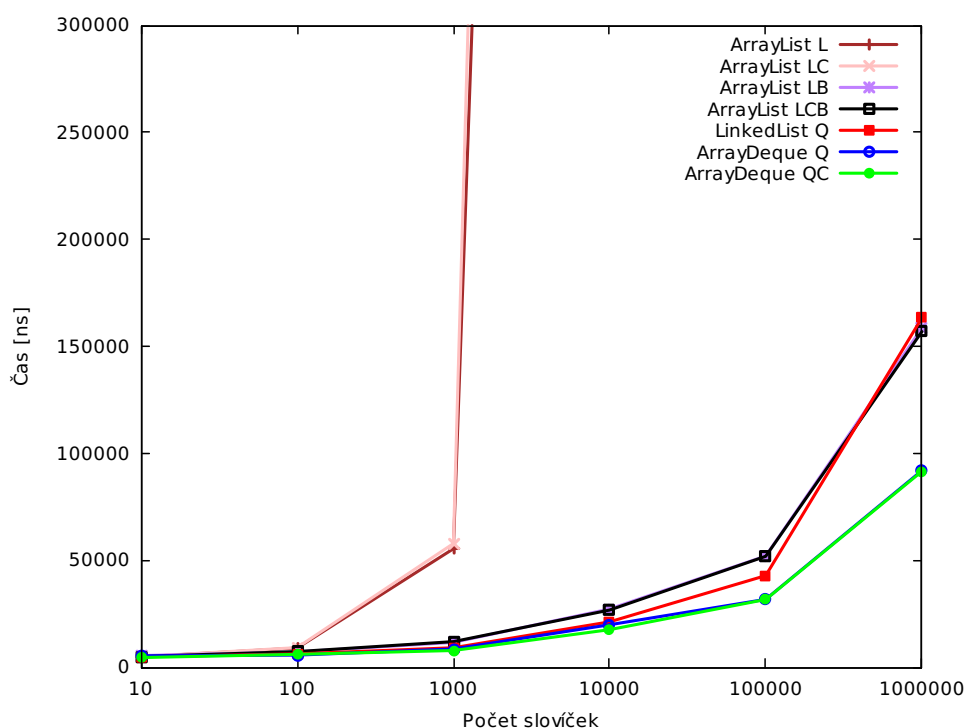
Paměťová složitost zůstává na logické složitosti $\mathcal{O}(n)$, kde n je počet slovíček pro překlad.

3.3.2.3 Záznam postupu

U každé lekce se drží kromě jiného i počet slovíček, které uživatel ve zkoušení již zodpověděl správně. Vzhledem k možnosti zkoušení z více aktuálně zvolených lekcí a možnosti opakovat zkoušení ihned by nebylo snadné tuto informaci udržovat a aktualizování databáze po každé odpovědi by nebylo efektivní. Proto jsem zvolil postup uložení unikátního identifikátoru lekce jako klíč a počet správně zodpovězených slovíček jako hodnotu. Slovíčko má v sobě uložen unikátní identifikátor jako celé číslo, které označuje lekci, do které patří. Při správně zodpovězeném slovíčku se tedy toto číslo získá a použije jako klíč a zvýší se hodnota, která je s daným klíčem spojená. Takovýto přístup je typický pro mapu.

Kromě speciálních implementací mapy jsou k dispozici tři hlavní, nejtypičtější **HashMap**, **ArrayMap** a **SparseArray**.

HashMap má oproti zbylým dvěma výhodu konstantního získání jakéhokoliv prvku, ale cenou za to je autoboxing klíče a v určitých případech pomalé hashování. V mém přístupu je třeba ukládat dva inty, ale díky autoboxingu



- L - Implementuje List
- Q - Implementuje Queue
- C - Předem stanovená kapacita
- B - Operace prováděné z konce

Obrázek 3.3: Graf rychlosti mazání z datových struktur

je nutné použít dva objekty Integer. Kromě těchto nevýhod je zde také vyšší režie, a to z důvodu kolize klíčů. Při kolizi klíčů, ale různých hodnotách se vytvoří nový objekt a původní objekt má ukazatel na něj. Vytváří se tím tak spojový seznam a získání prvku již není konstantní $\mathcal{O}(1)$. V mém případě k tomu však dojít nemůže.

ArrayMap používá dvě pole. Jedno pro hashe typu int a jedno pro hodnoty typu Object. Při vložení prvku se vypočítá z klíče hash, který se uloží na určené (dle seřazení) místo do prvního pole a uchová se jeho index. Tento index se vynásobí dvěma a získané číslo je indexem druhého pole pro uložení klíče a hodnoty o jedno vedle. Při získávání prvku se tak vypočítává hash, nalezne jeho index v prvním poli a jednoduchou aritmetikou se získá klíč i hodnota z druhého pole. V obou případech v části hashování se využívá binárního vyhledávání, takže celková složitost získání prvku je logaritmická $\mathcal{O}(\log n)$. Vzhledem k druhému poli typu Object se nedá vyhnout autoboxingu

ani u klíče, ani u hodnoty. Amortizované složitosti $\mathcal{O}(1)$ při vkládání do pole se dá vyhnout počátečním nastavením kapacity a posouvání prvků napravo od vkládaného hashe (složitost $\mathcal{O}(n)$) se dá vyhnout vkládáním již seřazených čísel.

`SparseArray`, zde konkrétně jeho podmnožina `SparseIntArray` je založená na podobném principu jako `ArrayMap`. Využívá pouze dvě pole pro klíče a hodnoty. Obě pole jsou typu `int`, což je nejmenší datový typ použitelný pro můj případ. Z toho také vyplývá, že se jeho použitím vyhnu autoboxingu a tedy ukládám čísla do nejprimitivnějších typů. Na rozdíl od struktury `ArrayMap` se nevytváří žádný hash, ale ukládá se klíč do prvního pole s tím, že stejný index obsahuje hodnotu v poli druhém. Klíče jsou seřazené a tak se při získávání indexu používá opět binární vyhledávání, což způsobí, že složitost přidávání i získávání je opět logaritmická $\mathcal{O}(\log n)$. Amortizované složitosti $\mathcal{O}(1)$ při vkládání do pole se dá opět vyhnout počátečním nastavením kapacity a posouvání prvků napravo od vkládaného prvku (složitost $\mathcal{O}(n)$) se opět dá vyhnout vkládáním již seřazených čísel.

Při analýze zdrojových kódů Javy jsem si naivně spočítal velikost jednotlivých struktur pro 1 000 prvků při použití celých čísel u klíčů i hodnot. Velikost `HashMap` činí přes 72 000 bytů, velikost `ArrayMap` činí přes 36 000 bytů a `SparseIntArray` lehce přes 8 000 bytů. Zde je využití mapy velice jednoduché a zároveň je nutné šetřit paměť. Je tedy jasné, že lze zvolit jediné `SparseIntArray`. Rychlost na tak malých datech by měla být sice identická pro všechny tři implementace mapy, ale velikost je významně nižší pro `SparseIntArray`. Využití má po každé správně zodpovězené otázce časovou složitost $\mathcal{O}(\log l)$ a celkovou paměťovou složitost $\mathcal{O}(l)$, kde l je počet aktuálně zvolených lekcí.

3.4 Databáze a vyhledávání

Spojení s SQLite databází zajišťuje třída `SQLiteOpenHelper`. Ta má čtyři důležité vlastnosti. První je název databáze, který se použije jako název souboru, ve které je databáze uložena. Druhou vlastností je celé číslo označující verzi databáze. Další dvě vlastnosti jsou dvě funkce, které se volají dle verze databáze. Pokud se číslo zvýšilo, volá se funkce `onUpgrade()`. Při snížení čísla se zavolá funkce `onDowngrade()`. Pouze funkce `onUpgrade()` je povinná a je na vývojáři, aby napsal algoritmus pro přenesení dat či změnu struktury ze staré verze do nové či naopak.

3.4.1 Primární tabulky

Pro základní funkcionalitu aplikace jsou třeba dvě tabulky. První tabulka shromažďuje lekce a druhá tabulka obsahuje jednotlivá slovíčka. Názvy tabulek odpovídají anglickým synonymům.

3.4.1.1 Struktura tabulky Lessons

- id **int** - unikátní identifikátor s primárním klíčem
- wordCount **int** - počet slovíček v lekci
- wordPassedCount **int** - počet správně zodpovězených slovíček v lekci
- lessonName **text** - název lekce
- selected **int** - zda je lekce vybrána jako aktuální (hodnoty 0/1)
- origin **text** - název souboru z kterého se lekce importovala
- tested **int** - kolikrát byla tato lekce dokončena ve zkoušení
- imported **datetime** - datum a čas importování lekce

3.4.1.2 Struktura tabulky Words

- id **int** - unikátní identifikátor s primárním klíčem
- lessonId **int** - cizí klíč do tabulky Lessons pro spárování s lekcí
- czech **text** - slovíčko v českém jazyce
- english **text** - slovíčko v anglickém jazyce
- failCount **int** - kolikrát bylo slovíčko špatně zodpovězeno při zkoušení
- state **int** - status slovíčka (hodnoty 0/1/2/3/6/7)

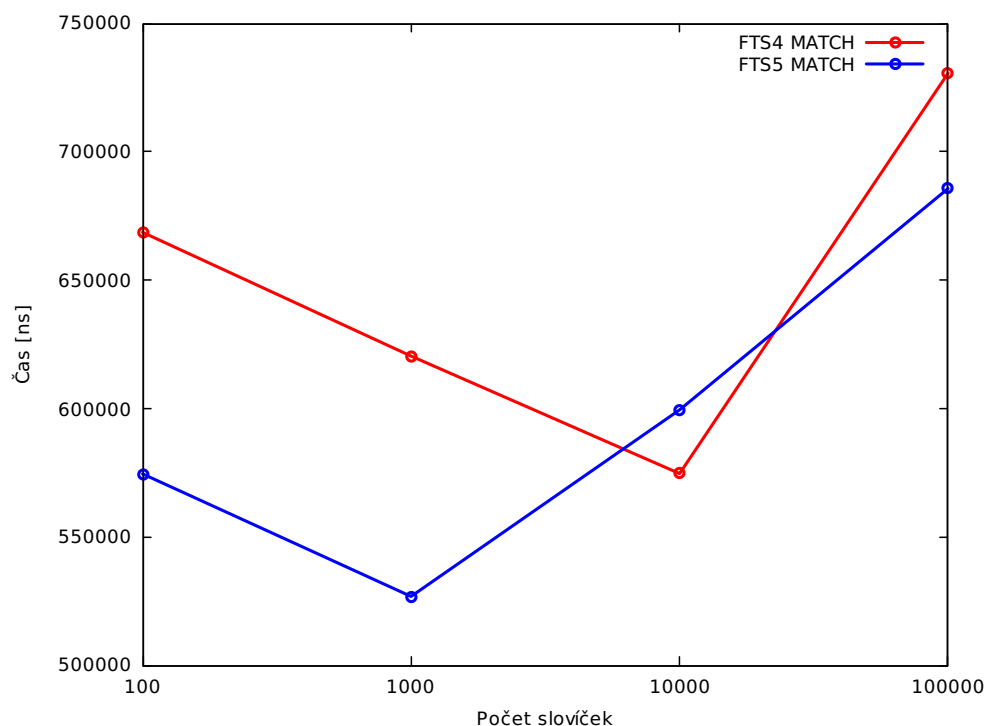
Status nabývá hodnot dle tří bitů. První bit značí učení, druhý zkoušení a třetí značí, zda bylo správně zodpovězeno při zkoušení.

3.4.2 FTS tabulka

Pro full-textové vyhledávání je nezbytná virtuální tabulka s modulem fts. Virtuální tabulka je napojena na klasickou tabulku Words a obsahuje tak ukazatel na ni. Tabulka je vytvořena příkazem `CREATE VIRTUAL TABLE selected USING fts4(content='words', czech, english)`. Je zřejmé, že index se vytváří pouze na sloupcích s českým a anglickým slovíčkem. Kromě těchto explicitních sloupců obsahuje virtuální tabulka implicitně důležitý sloupec **docid**, který má stejnou hodnotu jako primární klíč v originální klasické tabulce.

Problémem full-textového vyhledávání je jiný princip hledání. Zatímco u klasických vyhledávacích algoritmů nebo databázového `LIKE` můžu vyhledávat podřetězec celého textu, tak u `MATCH` je to jiné. Podřetězec lze vyhledat pouze se známým začátkem. Jinak řečeno, zástupný znak na začátku vyhledávaného řetězce nemá žádný význam.

3. REALIZACE



Obrázek 3.4: Časové porovnání full-textových modulů SQLite

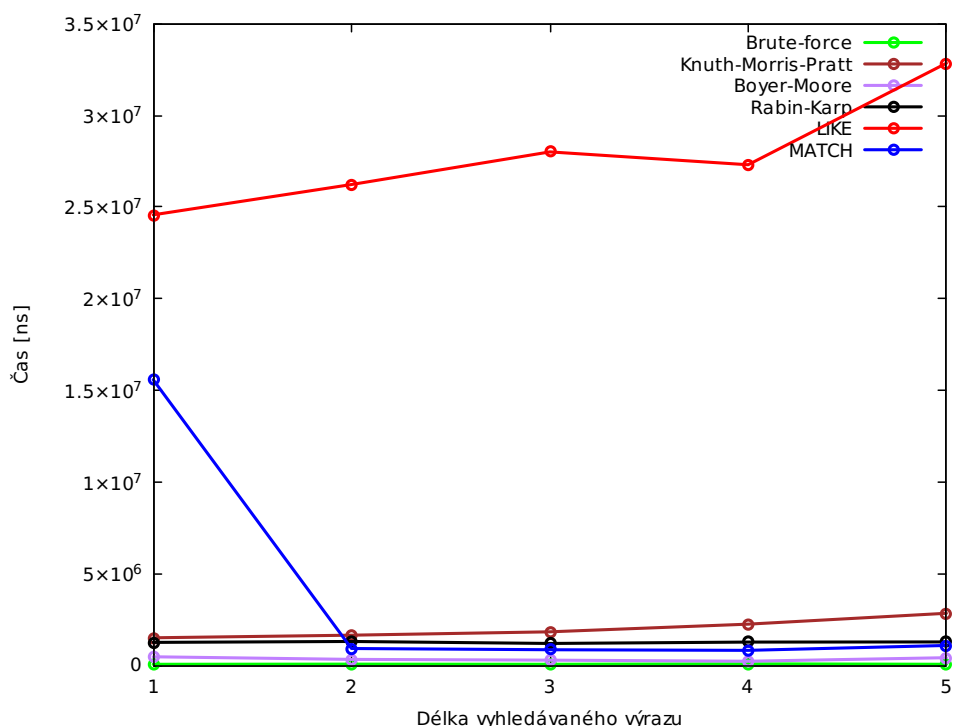
3.4.3 Rychlostní porovnání

Podle sekce 2.3 jsem implementoval několik algoritmů na vyhledávání a všechny je podrobil několika testům. Každý test jsem provedl několikrát a z výsledků vybral medián.

Vyhledávací algoritmy včetně brute-force algoritmu (funkce `indexOf()`) vyhledávají jakékoliv slovíčko či frázi, které obsahují zadaný výraz jako podřetězec. Databázová podmínka s `LIKE` je vybavená zástupnými znaky procenta na začátku i na konci zadaného výrazu, takže nachází stejná slova. `MATCH` nemá takovou variaci, takže obsahuje pouze zástupný znak na konci výrazu a při testech používám takové výrazy pro hledání, aby nalezená slova byla stejná jako u všech ostatních.

3.4.3.1 Nultý test

Nultý test se týká pouze full-textových modulů SQLite. Jelikož jde o stejnou funkci (`MATCH`) ve dvou různých verzích (4 a 5), tak není třeba v následujících testech zkoušet obě, ale jen jednu. Tu lepší. Z grafu 3.4 až na jeden výkyv jasně vyplývá, že lepší FTS je novější verze 5. V následujících testech a grafech tedy název `MATCH` přisuzuji nejnovější verzi FTS5.



Obrázek 3.5: Časové porovnání vyhledávacích algoritmů

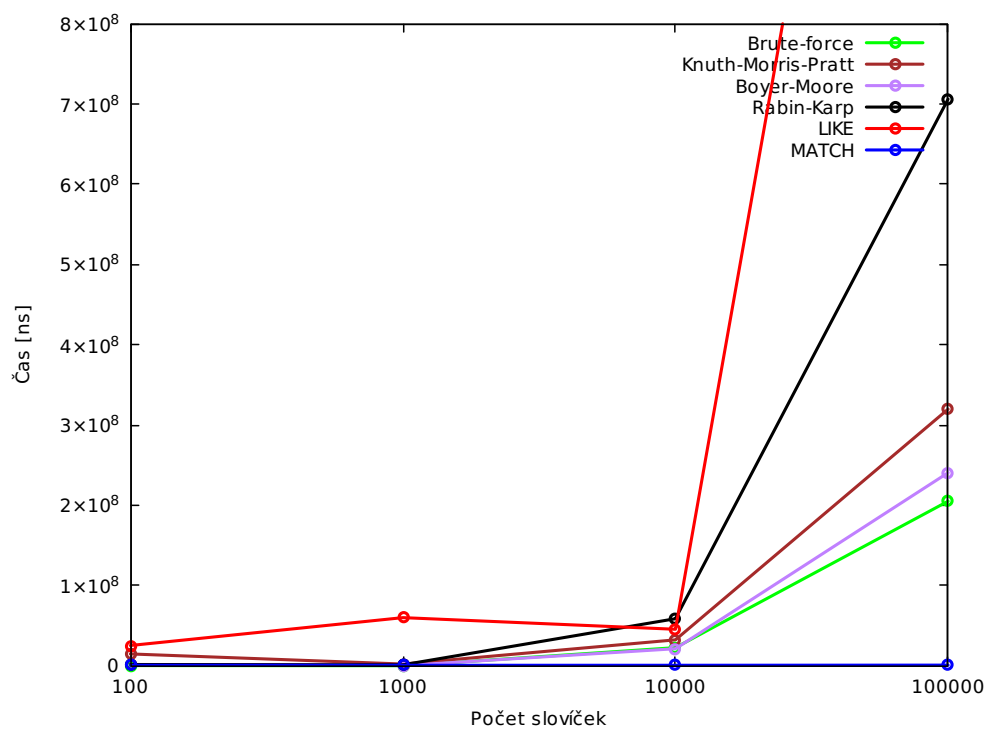
3.4.3.2 První test

První test se týká nejnepřítěžnějšího použití vyhledávání ve slovníku - vyhledávání v pár desítkách slovíček. Na grafu 3.5 lze vidět, že LIKE časově zaostává daleko za ostatními algoritmy, přestože se jedná o zlomky vteřiny. MATCH má špatný začátek při jediném znaku, ale od druhého znaku vyhledávaného výrazu je srovnatelný s ostatními. Ovšem jasným vítězem z pohledu časové náročnosti je zde obyčejný brute-force přístup.

3.4.3.3 Druhý test

Druhý test je zaměřený na počet slovíček, ve kterých se vyhledává. Vyhledávaný výraz má pevně stanovenou délku 4 znaky a počet slovíček se pohybuje od 100 až po 100 000, což 10x přesahuje zadání této práce. Na grafu 3.6 je patrné, že ani v tomto typu testu nemůže LIKE uspět. Oproti tomu MATCH se drží na konstantní rychlosti, ve které mu konkuruje pouze brute-force a výkyvově Boyer-Moore s lepšími časy do 1 000 slovíček. Tyto detailní časy lze zjistit z tabulky 3.2.

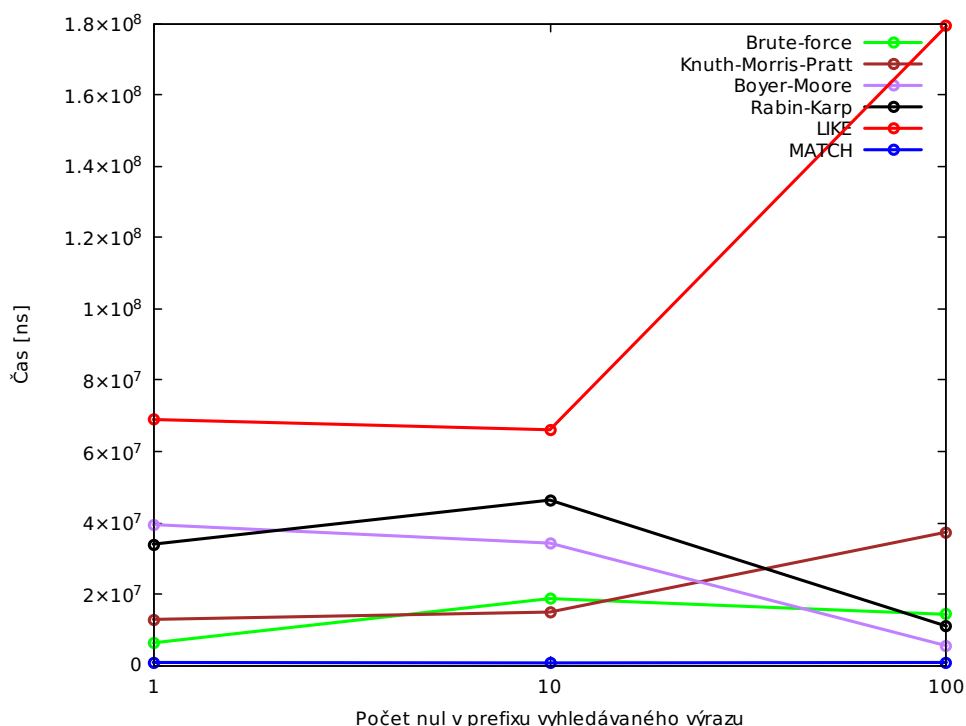
3. REALIZACE



Obrázek 3.6: Časové porovnání vyhledávacích algoritmů

Tabulka 3.2: Časové porovnání vyhledávacích algoritmů

	100	1000	10000	100000
Brute-force	141718	16927	22554740	204978020
Knuth-Morris-Pratt	14670990	2068750	32230209	319792083
Boyer-Moore	1538645	170469	21111094	240595052
Rabin-Karp	1455052	1103541	58605313	706059271
LIKE	25212552	59860260	45476822	1965261973
MATCH	574636	526875	599583	685730



Obrázek 3.7: Časové porovnání vyhledávacích algoritmů

3.4.3.4 Třetí test

Třetí a poslední test je zaměřen na nejhorší možná data. Slovník v tomto testu obsahuje 1 000 slovíček, kde každé slovíčko je sestavené z jedničky a proměnlivého počtu nul jako prefixu. Regulárním výrazem lze popsat slovíčka takto: „ $0\{1,1000\}1$ “. Vyhledávaným výrazem je stejné slovíčko s proměnlivým počtem 0 v prefixu. V tomto testu by se měla ukázat síla vyhledávacích algoritmů, což lze z grafu 3.7 vyčíst. Jak je vidět, tak čím vyšší počet nul v prefixu, tím mají algoritmy (až na KMP) nižší časové nároky. Vyhledávání pomocí LIKE je opět nejpomalejší a vyhledávání pomocí MATCH je opět nejstálenější. Ačkoliv zde bylo ukázáno, že brute-force není zdaleka vždy nejlepší řešení, tak tento test má pramálo společného s běžným využitím mé aplikace.

3.4.3.5 Vyhodnocení

Z těchto testů je jasné, že výběr algoritmu pro vyhledání závisí na počtu slovíček, ve kterých se má hledat. Pokud jich je málo, tak nelze překonat obyčejný brute-force přístup. Nicméně, při počtu 1 000 slovíček a více je lepší použít full-textové vyhledávání databází. To s sebou ale nese dva kompromisy. Prvním je oželení vyhledávání výrazu uprostřed slova, například výraz „pata“ nenalezne slovíčko „lopata“. Druhým kompromisem je vytváření samotného

3. REALIZACE

indexu. V sekci 2.3.6.2 jsem zmiňoval nutnost použít příkaz ke smazání a vytvoření celého indexu při každé změně originální tabulky nebo před každým vyhledáváním. Ovšem pokud se zvolí možnost vytváření indexu před vyhledáváním (například při kliknutí do políčka hledání) a slovíček je mnoho, tak už se to dá považovat za předzpracování, které nějakou dobu trvá. Ačkoliv taková doba není zanedbatelná, tak nedosahuje času vyhledávání pomocí LIKE.

Testování

4.1 Testování vývojářem

Během realizace jsem častokrát testoval, zda jsou algoritmy dostatečně rychlé a zda je vždy dostatek heap paměti. Kromě tohoto testování jsem musel testovat také funkčnost aplikace jako takové.

4.1.1 Základní testovací zařízení

Pro nejdůležitější testování je potřeba používat fyzická zařízení. Čím rozdílnější zařízení a s větším rozpětím verzí OS Android, tím větší šance na kompatibilitu celého spektra zařízení potenciálních zákazníků.

4.1.1.1 LG Nexus 5

Tento telefon je mým hlavním testovacím zařízením při vývoji. Důvodem je aktuálnost obsažené verze OS Android a také dostatečný výkon. Je to zároveň můj nynější vlastní telefon, který používám v běžném životě.

Verze OS Android: 6.0.1

Velikost displeje: 4.95"

Rozlišení: 1920 x 1080 px

RAM: 2 GB

Procesor: Snapdragon 800

4.1.1.2 HTC Desire

Tento telefon jsem použil jako vedlejší testovací zařízení pro jeho staří a menší výkonnost. Důvody, proč jej využít k testování, jsou vyzkoušení zpětné kompa-

4. TESTOVÁNÍ

tibility, použitelnosti a disponovatelnost externí SD kartou, pro kterou hlavní testovací zařízení nemá slot.

Verze OS Android: 4.4

Velikost displeje: 3.7"

Rozlišení: 800 x 480 px

RAM: 512 MB

Procesor: Snapdragon S1

4.1.1.3 Samsung Galaxy S5 mini

Tento telefon byl používán pouze v jedné části implementace a její testování z důvodu variace externích pamětí. Kromě klasického slotu pro externí SD kartu označovanou jako sekundární externí úložiště obsahuje tento telefon také emulovanou externí SD kartu označovanou jako primární externí úložiště.

Verze OS Android: 4.4.2

Velikost displeje: 4.5"

Rozlišení: 1280 x 720 px

RAM: 1,5 GB

Procesor: Cortex-A7

4.2 Testování uživateli

Uživatelé testovali jednu z posledních verzí, hlavně z důvodu dostatečného počtu dat z výkonnostních testů. Kromě těchto naměřených dat jsem byl zároveň otevřen zpětné vazbě, co se použitelnosti aplikace týče.

4.2.0.1 Samsung Galaxy S5

Tento telefon patří vedoucímu této práce, který testoval verzi určenou pro obchod Google play, jehož prostřednictvím byla aplikace a její aktualizace distribuovány do tohoto zařízení.

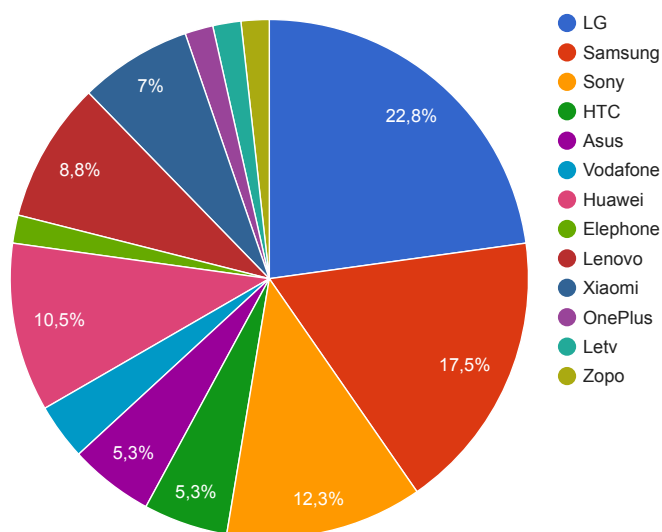
Verze OS Android: 5.0

Velikost displeje: 5.1"

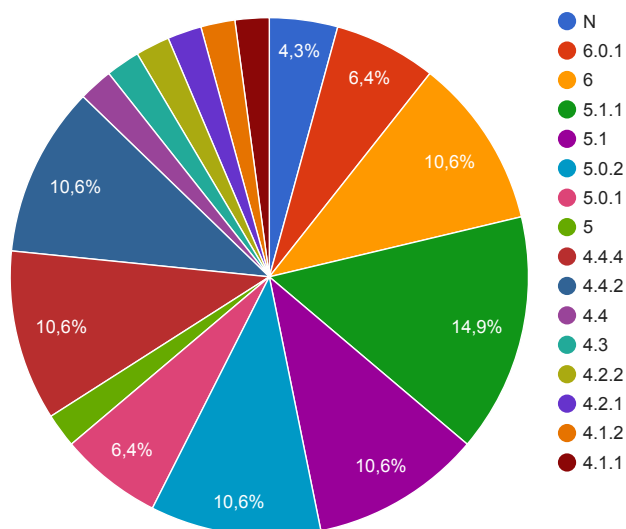
Rozlišení: 1920 x 1080 px

RAM: 2 GB

Procesor: Snapdragon 801



Obrázek 4.1: Znázorněné rozpětí značek testovacích zařízení

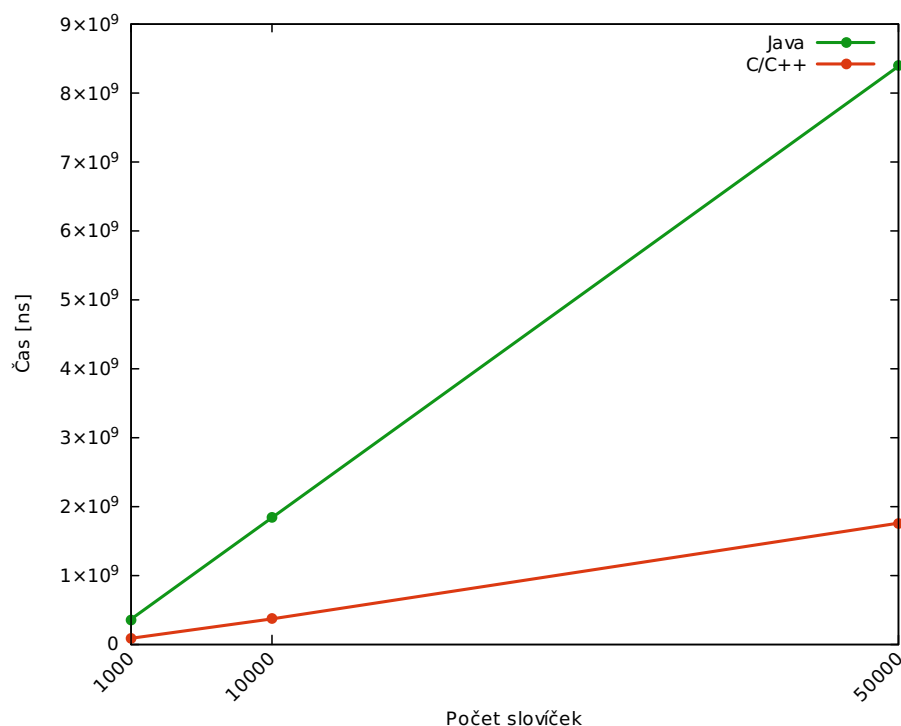


Obrázek 4.2: Znázorněné rozpětí verzí OS Android na testovacích zařízeních

4.2.1 Souhrn testovacích zařízení

Dle svého nejlepšího úsudku jsem odstranil identická zařízení a do grafu 4.1 a 4.2 jsem zaznamenal rozpětí značek mobilních zařízení a verzí OS Android. Dohromady jsem získal data z 57 (fyzicky) různých zařízení a jejich kompletní seznam lze najít v elektronické příloze.

4. TESTOVÁNÍ



Obrázek 4.3: Časové porovnání Javy a C/C++ dle velikosti dat

4.2.1.1 Závislost na velikosti dat

V sekci 3.2.2 jsem zmínil, že není třeba přidávat k benchmarku testy s různou velikostí dat, jelikož se ukázalo, že čas je přímo úměrný velikosti dat a C/C++ bylo vždy rychlejší. Jednoduché měření jsem zaznamenal do grafu 4.3. Jedná se o import řádek po řádku v Javě a C/C++ na třech různých objemech dat.

4.2.2 Výsledky výkonostních testů

Na rozdíl od testovacích zařízení jsem žádná naměřená data nemazal, protože každé měření mohlo být odesláno pouze jednou a každé měření je relevantní. Odeslaných měření bylo 69 v době zpracovávání. Pro všechna data jsem spočítal aritmetický průměr, který lze najít v grafech 4.4 a 4.5. Lze z nich snadno vyčíst, že souhrn úkonů, které jsem měřil, je v C/C++ mnohonásobně rychlejší. Zajímavější ovšem je, že rychlost těchto úkonů je vyšší na externích kartách než u interní paměti. Přisuzuji to zejména odbornější komunitě testerů, kteří investují peníze do rychlejších karet.

Nicméně aritmetický průměr je velice náchylný na extrém, které dle dat občas nastaly a tak jsem vypočítal také medián, který lze nalézt v grafech 4.6 a 4.7. V uvedených grafech se již rozdíl mezi externí a interní pamětí srovnal a blíží se tak logickému úsudku, že interní paměť by měla být rychlejší.

Tabulka 4.1: Časové porovnání vyhledávacích algoritmů v ms

Aritmetický průměr				
	Java		C/C++	
	int	ext	int	ext
Řádek po řádku	24235.907	20118.190	1542.510	2134.054
Buffer 32	15771.369	12249.773	1478.184	1193.381
Buffer 256	16338.141	12185.127	1384.019	1234.528
Buffer 2048	22906.560	12024.409	1297.821	1194.481
Buffer 16384	25169.096	12241.351	1193.336	1233.146
Buffer 65536	19845.816	12047.187	1238.407	1282.957

Medián				
	Java		C/C++	
	int	ext	int	ext
Řádek po řádku	11735.077	11989.119	1318.202	2112.781
Buffer 32	9469.832	10102.030	1115.192	1195.908
Buffer 256	9650.603	10018.867	1105.126	1191.754
Buffer 2048	9930.993	9558.645	1168.625	1196.392
Buffer 16384	9396.027	10029.834	1140.699	1221.947
Buffer 65536	9678.773	9756.814	1184.692	1230.734

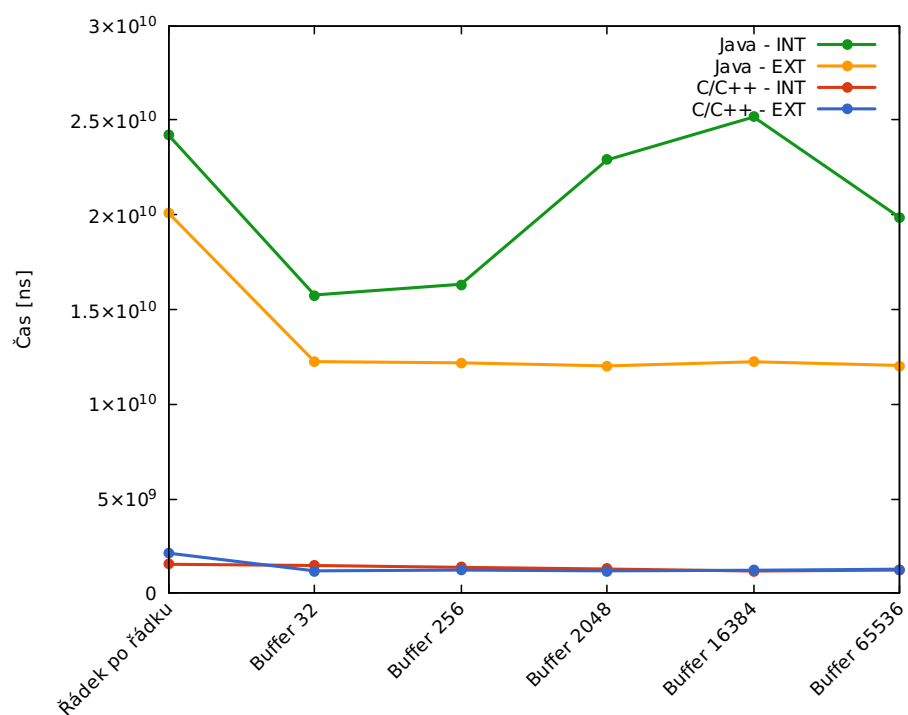
Pro detailnější pohled na znázorněná čísla je přidána tabulka 4.1. Všechna data lze najít v elektronické příloze.

4.2.2.1 Vyhodnocení

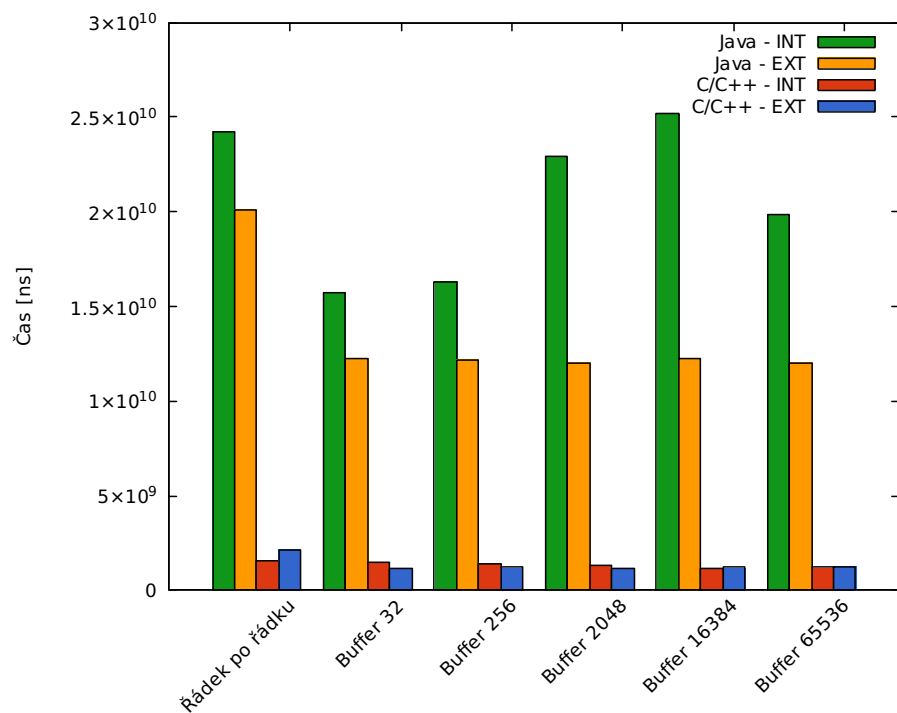
Z výše uvedených výsledků je jasné, že C/C++ je mnohonásobně rychlejší než konvenční Java. Aplikace pro OS Android nelze napsat celá v C/C++, protože samotné UI běží v ART, ale pokud aplikace zpracovává nějaká data, tak je jistě správná cesta alespoň vyzkoušet, zda-li to nebude rychlejší v C/C++. Obecně lze říci, že když se vývojář dostane do fáze, kdy je vše již plně optimalizováno, ale takzvané „úzké hrdlo“ (bottleneck) je stále příliš pomalé, tak je třeba jej zkusit implementovat v C/C++.

O tom, proč tomu tak je, lze polemizovat. Dohady mezi stranami jazyků Java a C/C++ byly vždy. Nicméně konkrétní fakta jsou, že Java kompiluje do bytekódu a následně do nativního kódu, kdežto C/C++ rovnou do nativního kódu. C/C++ také není limitováno heap pamětí a nepotřebuje Garbage collector.

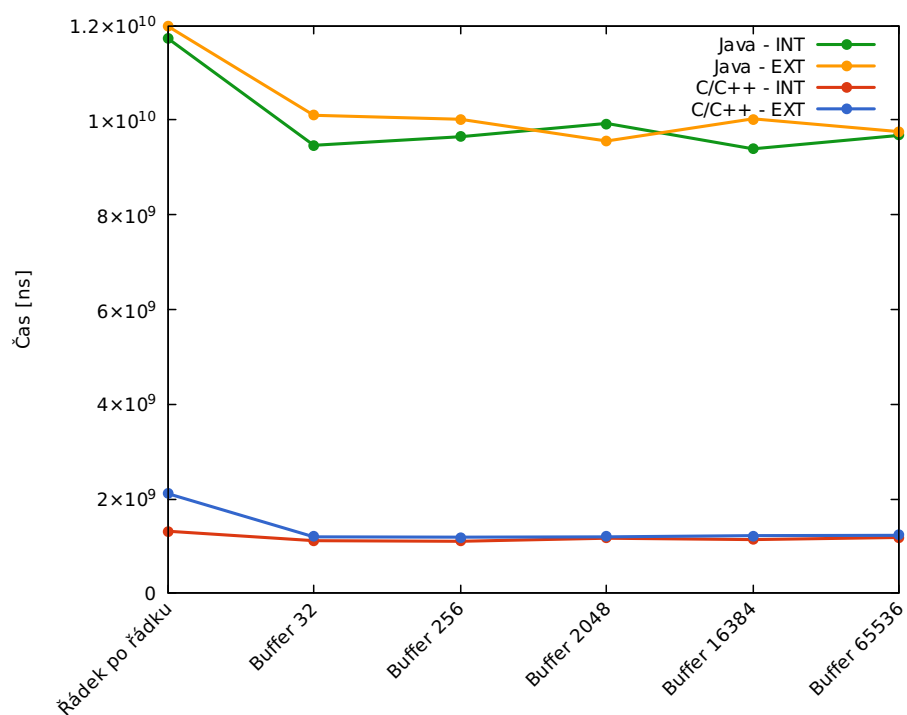
4. TESTOVÁNÍ



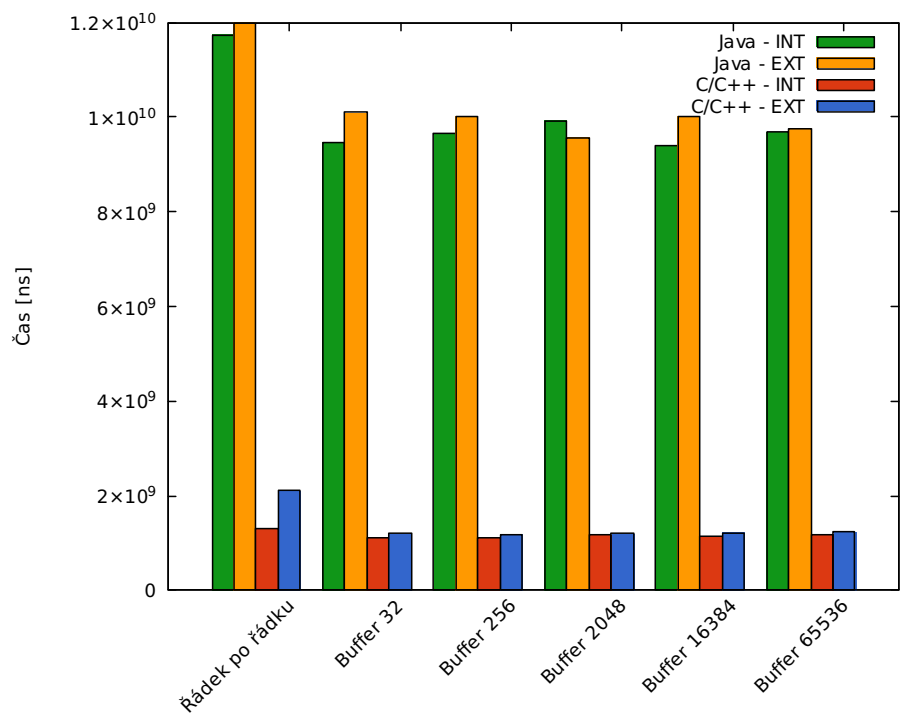
Obrázek 4.4: Aritmetický průměr výkonostních testů ve spojnicovém grafu



Obrázek 4.5: Aritmetický průměr výkonostních testů ve sloupcovém grafu



Obrázek 4.6: Medián výkonostních testů ve spojnicovém grafu



Obrázek 4.7: Medián výkonostních testů ve sloupcovém grafu

Další rozvoj aplikace

5.1 Krátkodobé cíle

Aplikace byla psána zejména s cílem vyzkoušení a změření různých algoritmů a metod vývoje. Jako taková má jistě daleko do bezchybné aplikace a z hlediska softwarového inženýrství má mnoho nedostatků ohledně použitelnosti.

Aplikace by se jistě mohla zaměřit na plné využívání C/C++, u kterého je třeba doplnit funkce, které by zpříjemnily uživatelské používání aplikace. Další změnou by mohl být formát souboru, který nyní využívá znak rovná se (=) pro oddělení dvojice slovíček a není tedy možné tento znak použít ve slovíčku. Vylepšením by mohlo být také odstranění omezení na angličtinu s češtinou a přidání dalších jazyků.

5.2 Dlouhodobé cíle

Aplikace vyžaduje OS Android v minimální verzi 4.1.x a jako taková není přístupná na mnoha zařízeních. Aplikace by sice ztratila na jednoduchosti, ale jistě je třeba přidat několik bloků kódu, které by se spouštěly pro starší, méně výkonná zařízení. Stejně tak by se musel značně omezit počet slovíček, který lze zpracovávat, jelikož starší zařízení nemají tolik přístupné paměti RAM, natož přidělené heap paměti.

Dalším dlouhodobým cílem je určitě zkoušení nových a lepších algoritmů. S každou verzí NDK se může objevit nová verze kompilátoru, opět rychlejší. S vývojem fyzických zařízení přichází i větší velikost paměti RAM a tak by časem mohl být bez obav zprovozněn lepší vyhledávací algoritmus obsahující paměťově náročné předzpracování.

Závěr

Cílem této bakalářské práce bylo vytvoření aplikace pro analýzu výkonnosti algoritmů a vyzkoušení nekonvenčních metod programování pro OS Android. Aplikace byla navržena jako správce anglické slovní zásoby.

Primárně jsem se zabýval návrhem a vyzkoušením efektivních algoritmů pro zajištění funkčnosti a stability aplikace při zpracovávání velkého množství slovíček. Zpracováváním se rozumí import vlastních slovíček, jejich správa a procvičování. Správa zahrnuje zobrazení slovníku, statistiky, mazání, sdílení a export.

Následně bylo nutné implementovat část aplikace nekonvenční metodou v jazyce C/C++. Konvenční programování pro OS Android se provádí v jazyce Java. Konečná aplikace obsahuje výkonnostní test porovnávající náročnou část aplikace v různých jazycích a implementacích. Tuto aplikaci jsem rozšířil mezi testovací komunitu ve formě beta testování a získal jsem zpětnou vazbu ve formě naměřených časů z výkonnostních testů.

Aplikace je nyní umístěna v obchodě Google play a je volně ke stažení pro veřejnost. Vyhledatelná je pod názvem „Správce slovní zásoby“ s mým jménem uvedeným jako autor. Také je přístupná skrze [tento odkaz](#) do obchodu Google play.

Literatura

- [1] Gartner: Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015 [online]. [cit. 2016-04-23]. Dostupné z: <http://www.gartner.com/newsroom/id/3215217>
- [2] Android Developers: Dashboards [online]. [cit. 2016-04-23]. Dostupné z: <http://developer.android.com/about/dashboards/index.html>
- [3] Tutorialspoint: Android - Architecture [online]. [cit. 2016-04-24]. Dostupné z: http://www.tutorialspoint.com/android/android_architecture.htm
- [4] Android Source: Android Interfaces and Architecture [online]. [cit. 2016-04-24]. Dostupné z: <https://source.android.com/devices/>
- [5] Google: Git at Google - kernel/common [online]. [cit. 2016-04-24]. Dostupné z: <https://android.googlesource.com/kernel/common/+refs>
- [6] Android Developers: Android Studio [online]. [cit. 2016-04-24]. Dostupné z: <http://developer.android.com/sdk/index.html>
- [7] Android Developers: Storage Options [online]. [cit. 2016-04-24]. Dostupné z: <http://developer.android.com/guide/topics/data/data-storage.html>
- [8] Google: Material design guidelines [online]. [cit. 2016-05-15]. Dostupné z: <https://www.google.com/design/spec/material-design/>
- [9] MAYEROVÁ, E.: *Aplikace pro zkoušení slovní zásoby na platformě Android*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.
- [10] Marko GARGENTA: Using NDK for Performance: Dalvik vs. Native [online]. [cit. 2016-05-11]. Dostupné z: https://newcircle.com/s/post/96/using_ndk_for_performance_dalvik_versus_native

- [11] SQLite: Database Speed Comparison [online]. [cit. 2016-05-09]. Dostupné z: <https://www.sqlite.org/speed.html>
- [12] Android Developers: Android NDK [online]. [cit. 2016-04-23]S. Dostupné z: <http://developer.android.com/tools/sdk/ndk/index.html>
- [13] Google: Android NDK [online]. [cit. 2016-04-23]. Dostupné z: http://developer.android.com/ndk/downloads/revision_history.html
- [14] Android Developers: Application.mk [online]. [cit. 2016-05-11]. Dostupné z: http://developer.android.com/ndk/guides/application_mk.html
- [15] SQLite: SQLite Is Public Domain [online]. [cit. 2016-05-15]. Dostupné z: <https://www.sqlite.org/copyright.html>
- [16] Google: JNI Tips [online]. [cit. 2016-05-15]. Dostupné z: <http://developer.android.com/training/articles/perf-jni.html>
- [17] Byron KIOURTZOGLU: Java Best Practices – String performance and Exact String Matching [online]. [cit. 2016-05-06]. Dostupné z: <https://www.javacodegeeks.com/2010/09/string-performance-exact-string.html>
- [18] Christian CHARRAS a Thierry LECROQ: Exact string matching algorithms [online]. [cit. 2016-05-06]. Dostupné z: <http://www-igm.univ-mlv.fr/~lecroq/string/>
- [19] SQLite: SQLite FTS3 and FTS4 Extensions [online]. [cit. 2016-05-07]. Dostupné z: <https://www.sqlite.org/fts3.html>
- [20] Java: Java Docs - ArrayDeque [online]. [cit. 2016-05-02]. Dostupné z: <https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html>

Seznam použitých zkratk

- OS** Operační systém
- API** Application programming interface
- IPC** Inter-process communication
- SSL** Secure sockets layer
- SQL** Structured query language
- DVM** Dalvik virtual machine
- JVM** Java virtual machine
- ART** Android runtime
- JIT** Just-in-time
- AOT** Ahead-of-time
- ADT** Android development tools
- UI** User interface
- SDK** Software development kit
- JDK** Java development kit
- NDK** Native development kit
- JNI** Java native interface
- USB** Universal serial bus
- ADB** Android debug bridge
- RAM** Random access memory

A. SEZNAM POUŽITÝCH ZKRATEK

ABI Application binary interface

MB Megabyte

GB Gigabyte

KMP Knuth-Morris-Pratt

BM Boyer-Moore

FTS Full-text search

Obsah přiloženého CD

readme.txt	stručný popis obsahu CD
apk	adresář se spustitelnou formou implementace
src		
_ impl	zdrojové kódy implementace
_ thesis	zdrojová forma práce ve formátu \LaTeX
text	text práce
_ thesis.pdf	text práce ve formátu PDF
attachments	adresář s přílohami
_ data.csv	souhrn naměřených dat
_ devices.csv	souhrn testovacích zařízení
_ types.csv	souhrn typů testů