VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO PROFILOVÁNÍ DATOVÝCH STRUKTUR PROGRAMŮ C/C++

DATA STRUCTURES PROFILING LIBRARY FOR C/C++ PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JIŘÍ PAVELA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ FIEDOR**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/20045/2016/xpavel32

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Pavela Jiří**
Obor: Informační technologie
Téma: **Knihovna pro profilování datových struktur programů C/C++**
Library for Profiling of Data Structures of C/C++ Programs
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Seznamte se s nástroji a přístupy pro profilování času a spotřeby paměti pro programy v C/C++ (MemTrack, HeapTrack, MemAnalyze).
2. Prostudujte dynamické datové struktury (seznamy, stromy, skip-list, red-black stromy, ...) a jejich asymptotické složitosti.
3. Navrhněte knihovnu pro snadné profilování datových struktur programů v C/C++.
4. Vytvořte nástroj pro přehlednou prezentaci získaných paměťových a časových profilů.
5. Nástroj demonstrujte na sadě alespoň pěti netriviálních datových struktur.

Literatura:

- Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, Charles E. Leiserson: Introduction to Algorithms, McGraw-Hill Higher Education, 2011, ISBN 0070131511.
- Domovské stránky projektu HeapTrack. URL: https://github.com/KDE/heaptrack

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Fiedor Tomáš, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Výkonnostní chyby mohou do velké míry negativně ovlivnit kvalitu vyvíjeného systému a v některých kritických odvětvích způsobit nevratné škody. K nalezení těchto chyb je obvykle využita profilace — v současnosti jedna z nejvíce rozšířených technik výkonnostní analýzy. Přestože existují rozšířené profilovací nástroje, tak většina z nich postrádá srozumitelný grafický výstup a schopnost detailnějšího studia složitostí algoritmů. Tato práce představuje nový profilovací nástroj zaměřený právě na automatizovaný odhad složitostí algoritmů a operací nad dynamickými datovými strukturami. Principem navrženého řešení je sběr statistických dat z běhu programu a následné provedení regresní analýzy s cílem nalézt model, který nejvhodněji reprezentuje složitost algoritmu. Výsledný prototyp byl podroben sérii experimentů, které vyhodnocují přesnost produkovaných výsledků, demonstrují praktická využití nástroje a názorně představují jeho grafický výstup.

Abstract

Performance bugs may greatly affect the quality of the system being developed and even cause irreversible damage in some critical sectors. Hence profiling — one of the currently most widespread technique of performance analysis — is usually applied to find the bugs. However, most of the current solutions commonly lack comprehensible graphical outputs and detailed analysis of algorithms in regard to their complexity. This thesis introduces a novel profiling tool which focuses on automatic estimation of complexity of dynamic data structures. The proposed approach collects statistical data out of program runs and uses regression analysis to find the most accurate model serving as an estimate of algorithmic complexity. The resulting prototype was subjected to a series of experiments that evaluate the accuracy of the results, demonstrate practical uses and illustrate the graphical output of the tool.

Klíčová slova

profilace, výkonnostní analýza, asymptotické složitosti, regresní analýza, algoritmy, dynamické datové struktury, C, C++

Keywords

profiling, performance analysis, asymptotic complexity, regression analysis, algorithms, dynamic data structures, C, C++

Citace

PAVELA, Jiří. *Knihovna pro profilování datových struktur programů C/C++*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Fiedor Tomáš.

Knihovna pro profilování datových struktur programů C/C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Fiedora. Další informace mi poskytla Mgr. Bc. Hana Pluháčková. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Pavela
16. května 2017

Poděkování

Rád bych tímto poděkoval vedoucímu práce Ing. Tomáši Fiedorovi za konzultace, rady, připomínky a odbornou pomoc s vývojem nástroje a textem práce. Dále bych rád poděkoval Mgr. Bc. Haně Pluháčkové za odborné konzultace k problematice regresní analýzy. V neposlední řadě bych chtěl vyjádřit poděkování své rodině a přítelkyni za podporu a trpělivost během doby, kdy jsem veškerou svou energii věnoval tvorbě této práce.

Obsah

Kapitola 1

Úvod

Algorithms + data structures = programs [23] od prof. Niklause Wirtha je publikace, která svým titulem vyjadřuje stále platné tvrzení, přestože byla vydána před více než 40 lety. A skutečně, budeme-li zkoumat jádro kernelu, informační systém nebo pomocný skript, vždy v nich ve výsledku nalezneme právě tyto dva základní stavební bloky. To, v čem se budou zkoumané programy lišit, je právě složitost jejich algoritmů a datových struktur.

Mluvíme-li o složitosti algoritmů, máme na mysli převážně jejich paměťové a časové nároky na provedení, může ale jít i o další výpočetní zdroje jako např. nároky na síťovou komunikaci nebo systémové operace. Při výběru vhodného algoritmu pro řešený problém se běžně zabýváme jeho teoretickou časovou složitostí, která se ale často může od praxe výrazně lišit. Právě kvůli této potenciální odlišnosti nás nezřídka zajímá, jakou časovou složitost skutečně vykazuje daný algoritmus v kontextu celého programu.

Tento problém částečně adresuje technika zvaná profilování — dynamická analýza pro hledání kritických míst v programu. V současnosti existují kvalitní profilovací nástroje, avšak jejich zaměření je převážně na program jakožto celek. Dokáží poskytnout informace o konkrétním provedeném běhu programu, postrádají však schopnost odhadnout změny chování jeho dílčích částí — datových struktur a algoritmů — pokud dojde ke spuštění nad jiným množstvím dat.

V tomto ohledu se práce snaží rozšířit současné možnosti profilace a poskytnout tak nástroj, který je schopen automatizované analýzy stavebních bloků C/C++ programu z pohledu časových složitostí. Důraz je kladen na odhad časové složitosti operací datových struktur a obecně tedy algoritmů pracujících nad proměnlivým počtem dat, mezi které patří například řadicí algoritmy. Součástí řešeného problému je i přehledná vizualizace výsledků, která usnadňuje jejich interpretaci uživatelem.

Kapitoly 2 a 3 se věnují teoretickému úvodu do oblasti složitostí a dynamických datových struktur. Mimo jiné je představen současný způsob odvození časových složitostí — teoretická analýza zdrojového kódu operace nebo algoritmu. V kapitole 4 je věnován prostor zběžnému představení nástroje Perun [12], který integruje výsledek této práce a jehož cílem je dále automatizovat současná profilační řešení.

Kapitoly 5, 6 a 7 přibližují samotné řešení problému sestávající ze sběru profilovacích dat, jejich analýzy a následné vizualizace. Každá z kapitol se věnuje jak analýze, tak návrhu a implementaci daného podproblému.

Na závěr práce je v kapitole 8 představeno několik experimentů, které ověřují funkčnost nástroje, demonstrují jeho možnosti a představují jeho případná praktická využití.

Kapitola 2

Časová složitost algoritmů

Tato kapitola vychází z Kapitol 2, 3 a 17 v [9] a stručně uvádí základní pojmy problematiky analýzy časových nároků algoritmů. Představuje nejpoužívanější způsoby odvození časové složitosti — asymptotickou a amortizovanou. U asymptotické složitosti se zpravidla zabýváme analýzou složitosti v nejhorším případě (tzv. *worst-case analysis*) a zběžně je popsána i složitost v průměrném (tzv. *average-case analysis*) a nejlepším (tzv. *best-case analysis*) případě.

V další části je pak popsána analýza tzv. amortizované složitosti, která na rozdíl od asymptotické analýzy umožňuje přesnější odhad horního ohraničení funkce složitosti. Jsou představeny dvě techniky amortizované analýzy — agregovaná analýza (tzv. *aggregate analysis*) a účetní metoda (tzv. *accounting method* nebo i *banker's view of amortization* [21]).

V sekci 2.2.5 je na příkladu algoritmu `insertion-sort` ilustrována technika odvození asymptotické složitosti. Obě techniky jsou prakticky ilustrovány na vybraných algoritmech v sekci 2.4.

2.1 Úvod do problematiky

Obecně lze dobu běhu programu vyjádřit jako funkci velikosti vstupních dat. Velikost vstupních dat je běžně značena písmenem n, jehož bližší význam závisí na typu úlohy (například počet prvků pole při jeho řazení). Samotná doba běhu programu je zpravidla reprezentována počtem kroků, neboli množstvím provedených primitivních operací stroje, kde každá operace trvá určitou dobu.

Abstrahujeme-li dobu skutečně potřebnou pro provedení operace a nahradíme ji konstantou (všechny operace tak trvají stejně dlouhý časový úsek), pak lze celkovou dobu běhu programu vyjádřit následovně. Je-li i-tý řádek kódu algoritmu proveden právě n-krát (počet provedení tedy závisí na velikosti vstupu) a každé jeho provedení vyžaduje c_i primitivních operací, pak provedení daného řádku kódu zvýší celkovou dobu běhu programu o hodnotu $c_i \cdot n$.

Řád růstu funkce (tzv. *order of growth***).** Při analýze složitosti často dochází k zanedbání veškerých konstant. Tyto konstanty nejsou pro asymptotickou složitost natolik podstatné, protože na celkovou dobu běhu programu nemají příliš velký vliv při dostatečně velkém n. Řád růstu funkce je nejvyšší mocnina n vyskytující se ve funkci složitosti a na celkovou dobu běhu algoritmu má největší vliv. Z tohoto důvodu jsou ostatní řády n z funkce zanedbány a složitost algoritmu je tak běžně charakterizována pouze řádem růstu. Jako

příklady častých řádů funkce lze uvést logaritmický ($\log n$; např. složitost vyhledání prvku v binárním vyhledávacím stromu), lineární (n; např. složitost vyhledání prvku v neseřazeném poli), linearitmické ($n \log n$; např. seřazení množiny prvků algoritmem `merge-sort`), kvadratický (n^2; např. seřazení množiny prvků algoritmem `insertion-sort`) nebo exponenciální (a^n; např. řešení problému obchodního cestujícího)[1].

2.2 Asymptotická složitost

Řád růstu funkce složitosti je často využíván pro charakterizaci a srovnávání jednotlivých algoritmů z hlediska jejich výpočetní a prostorové složitosti. Tento způsob srovnání však zanedbává konstanty a ostatní řády z funkce složitosti, což může způsobit ztrátu informace a reálný výkon algoritmů se přitom může v praxi výrazně lišit. Velikost vstupu algoritmu n proto musí být natolik velká, aby provedené zanedbání mělo velmi malý až téměř žádný vliv na výsledek. Standardní knihovna programovacího jazyka `C#` například implementuje ve svých řadicích metodách několik různých řadicích algoritmů a v závislosti na velikosti vstupních dat je automaticky zvolen ten nejvýhodnější algoritmus [17].

Studium algoritmů se vstupy natolik velikými, že pouze řád růstu je relevantní pro celkovou výpočetní složitost, označuje tzv. asymptotickou složitost algoritmu. Obecně tak dochází k analýze chování algoritmu při velikosti vstupu, který se limitně blíží k nekonečnu.

2.2.1 Asymptotická notace

Notace používaná pro popis asymptotické výpočetní složitosti je definována pomocí funkcí s definičním oborem celých nezáporných čísel ($\mathbb{N} = \{0, 1, 2, ...\}$). Zmíněný definiční obor je vhodný pro celočíselně udávanou velikost vstupu n, avšak existují i rozšíření využívající definiční obor reálných čísel. Samotná asymptotická notace definuje celkem 6 dalších notací, z nichž nejdůležitější jsou Θ-notace, $\mathcal{O}$-notace a Ω-notace, které definují množiny funkcí vstupu $f(n)$ v závislosti na funkci $g(n)$.

- Θ-notace je využívána pro popis složitosti v průměrném případě, blíže viz sekce 2.2.3.

$$\Theta(g(n)) = \{f(n) \mid \exists\, c_1, c_2, n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall\ n \leq n_0\} \tag{2.1}$$

  Tvrdíme, že $g(n)$ asymptoticky těsně ohraničuje (*asymptotically tight bound*) funkci $f(n)$. Pro všechna n větší než n_0 platí, že hodnota $f(n)$ je menší než $c_2 \cdot g(n)$ a větší než $c_1 \cdot g(n)$. Pokud takové konstanty c_1, c_2 a n_0 existují, tak funkce $f(n)$ patří do množiny $\Theta(g(n))$. Obrázek 2.1(a) graficky ilustruje Θ-notaci.

- $\mathcal{O}$-notace je používána pro popis složitosti v nejhorším případě, více v sekci 2.2.2

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists\, c, n_0 : 0 \leq f(n) \leq c \cdot g(n)\ \forall\ n \leq n_0\} \tag{2.2}$$

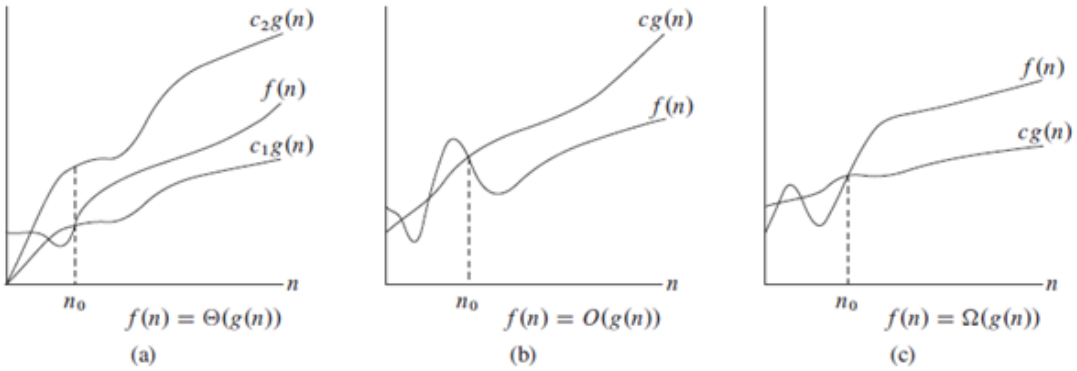
  Na rozdíl od Θ-notace, která popisuje ohraničení funkce zdola i shora, $\mathcal{O}$-notace definuje pouze asymptotické ohraničení shora (*asymptotic upper bound*). Pro všechny hodnoty n větší než n_0 platí, že hodnota $f(n)$ je stejná nebo menší než $c \cdot g(n)$. Stejně jako u Θ-notace, pokud existují vyhovující konstanty c a n_0, pak funkce $f(n)$ patří do množiny $\mathcal{O}(g(n))$. Notace známá rovněž jako „notace velké O“ (*Big O notation*[2]) je důležitá pro analýzu složitosti v nejhorším případě (sekce 2.2.2). Graficky znázorněna je $\mathcal{O}$-notace na obrázku 2.1(b).

[1] https://en.wikipedia.org/wiki/Time_complexity#Table_of_common_time_complexities
[2] https://en.wikipedia.org/wiki/Big_O_notation

- Ω-notace je využívána pro popis složitosti v nejlepším případě, viz sekce 2.2.4.

$$\Omega(g(n)) = \{f(n) \mid \exists\, c, n_0 : 0 \leq c \cdot g(n) \leq f(n)\ \forall\, n \leq n_0\} \tag{2.3}$$

  Ω-notace poskytuje stejně jako $\mathcal{O}$-notace pouze jedno ohraničení funkce $f(n)$, v tomto případě konkrétně asymptotické ohraničení zdola (*asymptotic lower bound*). Pro všechna n větší než n_0 platí, že hodnota funkce $f(n)$ je rovna nebo větší než $c \cdot g(n)$. Pokud takové konstanty c a n_0 existují pro danou funkci $f(n)$, pak $f(n)$ patří do množiny $\Omega(g(n))$. Ilustrace Ω-notace je na obrázku 2.1(c).



Obrázek 2.1: Grafické znázornění notací a) v průměrném (Θ), (b) nejhorším ($\mathcal{O}$) a c) nejlepším (Ω) případě [9].

Vzájemný vztah těchto tří notací lze vyjádřit následovně:

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n)) \tag{2.4}$$

2.2.2 Asymptotická složitost v nejhorším případě

Jedná se o nejčastěji používaný typ analýzy asymptotické složitosti, protože analýza nejhoršího možného případu poskytuje cennou informaci o maximální době běhu algoritmu v závislosti na velikosti vstupu. Analyzujeme algoritmus tak, že vždy musí provést maximální počet kroků pro získání požadovaného výsledku.

Popis asymptotické složitosti v nejhorším případě využívá O-notaci. Notace je využita proto, že funkce $c \cdot g(n)$, tvořící asymptotické ohraničení shora nad zkoumanou funkcí složitosti $f(n)$, formálně vyjadřuje nejhorší možný průběh funkce $f(n)$.

Příklad. Pro ilustraci lze uvést `insertion-sort` algoritmus, jehož nejhorší případ je opačně seřazené pole na vstupu. V tomto případě musí algoritmus provést vždy maximální počet kroků pro nalezení následujícího prvku, který má být vložen do již seřazené posloupnosti. Podrobnější analýzu asymptotické složitosti v nejhorším případě pro algoritmus `insertion-sort` lze nalézt v sekci 2.2.5.

2.2.3 Asymptotická složitost v průměrném případě

Možnou alternativou je analýza asymptotické složitosti v průměrném případě. Tento typ analýzy je obecně náročnější a využívá pravděpodobnostní analýzy (*probabilistic analysis*

pro výpočet doby běhu algoritmu. Nutnou podmínkou pro pravděpodobnostní analýzu je znalost rozložení vstupů. Pokud není rozložení vstupů známo, je nutné provést alespoň odhad. Tato podmínka je však u některých případů nesplnitelná a pak nelze analýzu provést.

Obecně dochází touto analýzou ke zjištění průměrné doby běhu přes všechny vstupy. Přínos analýzy pro znalost chování algoritmu se velmi liší v závislosti na zkoumaném problému. Zatímco u některých problémů je výsledek analýzy v průměrném případě stejný jako v nejhorším případě (co se týče řádu růstu), existují i algoritmy, jejichž složitost v nejhorším případě je odlišná od skutečně průměrné doby běhu algoritmu.

Příklad. Asymptotická časová složitost algoritmu `quick-sort` v nejhorším případě je kvadratická, avšak jeho časová složitost v průměrném případě je linearitmická. Tento rozdíl ve složitosti vzniká při výběru *pivotu*, což je hodnota, pomocí které je rozdělena množina řazených hodnot pro potřeby dalšího řazení. Při opakovaném výběru nevhodného pivotu (nejmenší nebo největší prvek množiny hodnot) tak algoritmus dosahuje časové složitosti $\mathcal{O}(n^2)$. Pokud je však hodnota pivotu volena vhodně (medián množiny prvků), nabývá časová složitost algoritmu hodnoty $\Theta(n \log n)$ [24].

2.2.4 Asymptotická složitost v nejlepším případě

Analýza asymptotické složitosti v nejlepším případě analyzuje takový vstup algoritmu, při kterém jsou časové nároky na provedení algoritmu nejmenší. Jako příklad lze uvést algoritmus řazení prvků v poli, jehož vstupem je již seřazené pole. Analýza složitosti v nejlepším případě neposkytuje informace důležité pro analýzu algoritmů a je zmíněna pouze pro úplnost. Formálně je vyjádřena pomocí Ω-notace a možné využití nalézá při zjišťování asymptoticky těsného ohraničení (viz sekce 2.2.1).

2.2.5 Demonstrace asymptotické analýzy

Analýza asymptotické složitosti v nejhorším případě je demonstrována na příkladu algoritmu `insertion-sort`, který je převzat z [9] (kapitola 2.2). Algoritmus včetně rozboru jednotlivých řádků kódu je vyobrazen na obrázku 2.2.

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*[A]	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

Obrázek 2.2: Algoritmus `insertion-sort` včetně ceny operací (sloupec *cost*) a počtu provedení (slupec *times*). Písmeno A značí vstupní pole k seřazení, n značí počet prvků v poli, t_j označuje počet provedení řádku algoritmu pro každou hodnotu j a ▷ označuje komentář ve zdrojovém kódu. [9].

Celkový čas běhu algoritmu $T(n)$ lze vyjádřit pomocí analýzy zdrojového kódu jako

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot \sum_{j=2}^{n} t_j + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot \sum_{j=2}^{n} (t_j - 1) + c_8 \cdot (n-1) \tag{2.5}$$

V nejhorším případě je pole na vstupu opačně seřazeno a prvek $A[j]$ je tak porovnáván s každým prvkem v již seřazené posloupnosti $A[1 \dots j-1]$. Z toho plyne

$$t_j = j : \forall\, j \in 2, 3, \dots, n \tag{2.6}$$

Potom lze vyjádřit

$$\sum_{j=2}^{n} j = \frac{n \cdot (n+1)}{2} - 1 \tag{2.7}$$

$$\sum_{j=2}^{n} (j-1) = \frac{n \cdot (n-1)}{2} \tag{2.8}$$

Při dosazení do rovnice 2.5 a její úpravě získáme následující vztah pro celkovou dobu výpočtu $T(n)$

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}) \cdot n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8) \tag{2.9}$$

Asymptotická složitost algoritmu v nejhorším případě tak nabývá výrazu an^2+bn+c. Po zanedbání konstant získáme výraz ve tvaru $n^2 + n$ a jeho řád růstu je proto n^2. Algoritmus tedy nabývá kvadratické časové složitosti v nejhorším případě ($\mathcal{O}(n^2)$). □

2.3 Amortizovaná složitost

Pojem *Amortizace* má původ ve finančním sektoru, kde vyjadřuje proces splácení dluhu po malých částkách během dlouhého časového intervalu [11]. Na poli informačních technologií byl pojem *amortizovaná časová složitost* poprvé použit R. E. Tarjanem [21], kde nabývá podobného významu.

Amortizovaná časová složitost označuje způsob analýzy časových nároků algoritmu, kdy jsou doby běhu jednotlivých operací v sekvenci váhované přes celou sekvenci operací. Při práci nad datovou strukturou je z pohledu doby běhu častěji důležitá sekvence provedených operací, než jednotlivé operace v sekvenci. Amortizovaná analýza může být využita pro zjištění, že přestože některé operace v sekvenci mohou být více výpočetně náročné, sekvence jako celek má daleko nižší průměrnou časovou složitost. Analýza asymptotické složitosti v nejhorším případě by však vycházela pouze z výpočetně nejnáročnější operace, čímž by vzniklo nereálně pesimistické horní ohraničení funkce složitosti [21].

Amortizovaná analýza, na rozdíl od asymptotické složitosti v průměrném případě, nepoužívá pro vyjádření průměrných výpočetních nároků pravděpodobnostní analýzu a odhady rozložení vstupů. Význam průměru je v tomto případě chápán více jako průměrné časové

nároky na každou operaci v nejhorším případě. Jedná se tedy stále o analýzu v nejhorším případě.

Agregační, účetní a potenciálové metody jsou tři nejběžnější techniky využívané při amortizované analýze a první dvě zmíněné budou zběžně popsány. Metoda potenciálů je principem velmi podobná účetní metodě, pouze používá jinou notaci. Detaily o metodě potenciálů lze nalézt v [21] a [9].

2.3.1 Agregační analýza

Agregační analýza je původem starší než amortizovaná analýza, v současnosti je však chápána více jako jedna z jejích forem [11]. Uvažujme pro různá n sekvenci obsahující n operací, kde celková doba běhu v nejhorším případě dosahuje hodnoty $T(n)$. Potom průměrná (amortizovaná) cena operace je dána výrazem $T(n)/n$. Důležitá je skutečnost, že tato cena byla zjištěna z analýzy v nejhorším případě.

Cena operace je stejná pro všechny operace v sekvenci, nezávisle na tom, že se zde mohou vyskytovat různé typy různě náročných operací, a proto tato metoda nemusí vést na přesný odhad složitosti. Účetní metoda, vylepšující agregační metodu, bere tuto skutečnost v potaz.

2.3.2 Účetní metoda

Na rozdíl od agregační analýzy, účetní metoda přiřazuje různým operacím různé ceny a navíc umožňuje některé operace nadhodnotit nebo podhodnotit oproti jejich skutečné ceně. Ohodnocení operace nazýváme *amortizovanou cenou*. Pokud amortizovaná cena $\widehat{c_i}$ i-té operace přesahuje skutečnou cenu c_i, pak každé provedení této operace přičte rozdíl $\widehat{c_i} - c_i$ k hodnotě *kreditu*, jehož počáteční hodnota je 0. Nastřádaný kredit pak může být využit k pokrytí ceny operace, jejíž cena $\widehat{c_i} < c_i$. Provedení operace tedy zvyšuje nebo snižuje hodnotu kreditu v závislosti na rozdílu cen $\widehat{c_i}$ a c_i, nebo nemá na uložený kredit žádný vliv v případě, že $\widehat{c_i} = c_i$.

R. E. Tarjan [21] definuje tento princip pomocí abstraktního modelu počítače fungujícího na mince (*coin-operated computer*). Vložením mince, která plní funkci kreditu, zajistíme běh počítače na určitou pevně danou dobu. Každý typ operace má přiřazen neměnný počet kreditů potřebných pro vykonání dané operace, čímž je vyjádřen amortizovaný čas operace. Smyslem je dokázat, že celá sekvence operací může být provedena během vyhrazeného času („zaplaceného“ kreditem, tedy vloženými mincemi) za předpokladu, že nevyužitý čas jednotlivých operací je možno využít pro následující operace.

Ohodnocení operací amortizovanou cenou musí splňovat jistá kritéria pro dosažení korektního výsledku. Celková amortizovaná cena sekvence operací musí být horním ohraničením skutečné ceny, aby bylo opravdu dosaženo průměrné ceny operace v nejhorším případě. Výsledná hodnota kreditu po provedení libovolné sekvence n tak musí být nezáporná a pro všechny sekvence n operací musí navíc platit nerovnice:

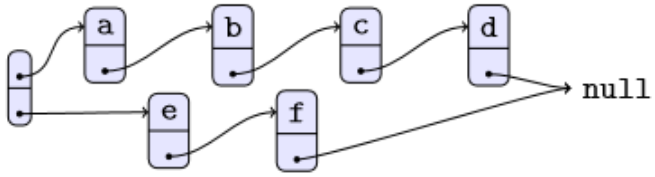
$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i \tag{2.10}$$

Lze provést i vypůjčení kreditů za účelem uhrazení nákladných operací, které mohou být prováděny na začátku sekvence. Vzniklý dluh ale musí být ve výsledku vždy celý uhrazen, aby byla zajištěna platnost nerovnice. Z praktického hlediska tedy lze v určitých částech sekvence operací pozorovat, že doba běhu přesahuje očekávaný amortizovaný čas, avšak tento rozdíl je vykompenzován ve zbylé části sekvence [21].

2.4 Demonstrační příklad

Analýzu amortizované složitosti a srovnání obou technik analýzy složitosti budeme demonstrovat na příkladu datové struktury funkcionální fronty (*Functional Queue*). Funkcionální fronta je tvořena dvěma jednosměrně vázanými seznamy (sekce 3.2.3), které simulují klasickou frontu (viz 3.2.2). Tento příklad je částečně převzat z [3].

Jeden ze seznamů simuluje přední část fronty (*head*) a druhý seznam její zadní část (*tail*). Při operaci vložení prvku do fronty (*enqueue*) dochází ke vložení prvku na první místo seznamu *tail*. U operace vyjmutí prvku z fronty (*dequeue*) dochází k odebrání prvního prvku seznamu *head*. V případě, že je seznam *head* prázdný, dochází k operaci reverzace nad seznamem *tail* a následným přiřazením hodnoty ukazatele na první prvek seznamu *head* na takto obrácený seznam. Ukazatel na první prvek seznamu *tail* je nastaven na hodnotu *NULL*, čímž reprezentuje prázdný seznam. Grafické znázornění funkcionální fronty je na obrázku 2.3.



Obrázek 2.3: Znázornění funkcionální fronty s obsahem [a, b, c, d, f, e], kde vrchní seznam funguje jako přední část fronty a spodní seznam jako její zadní část [3].

2.4.1 Asymptotická analýza

Algoritmy 1 a 2 obsahují pseudokódy operací vložení a odebrání prvku nad funkcionální frontou. Nad těmito algoritmy je provedena analýza asymptotické časové složitosti v nejhorším případě.

Algorithm 1 Pseudokód operace vložení prvku do funkcionální fronty.

Input:
Q ▷ The functional queue
x ▷ The element to enqueue

1: **function** ENQUEUE(Q, x)
2: $x.next \leftarrow Q.tail$
3: $Q.tail \leftarrow x$
4: **end function**

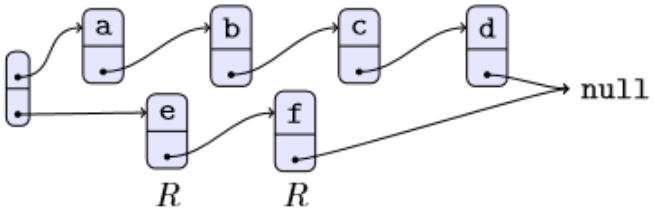
Při analýze asymptotické složitosti operací lze pozorovat, že operace vložení prvku má vždy konstantní čas. Operace odebrání prvku však může nabývat konstantní, nebo lineární časové složitosti. A to konkrétně v závislosti na tom, zda stačí pouze vyjmout první prvek ze seznamu *head*, nebo musí dojít k reverzaci celého seznamu *tail* pomocí cyklu. Asymptotická složitost v nejhorším případě tedy nabývá hodnoty $\mathcal{O}(n)$.

2.4.2 Amortizovaná analýza

Využitím amortizované analýzy složitosti je dosaženo lepší horní ohraničení operace *dequeue*. Amortizovaná cena operace vložení prvku nabývá hodnoty 2 jednotek zdroje ($2 \cdot R$), kde jedna jednotka bude využita na provedení samotné operace a druhá jednotka bude uschována na kredit. Cena operace odebrání prvku je 1 R a ten je bezezbytku využit na uvolnění prvku ze seznamu. Pokud při odebrání prvku dojde k potřebě provést reverzaci seznamu *tail*, je provedení této operace uhrazeno z uschovaných kreditů. Všechny prvky seznamu *tail* musely být nejprve vloženy operací *enqueue* a s každým prvkem tak lze asociovat právě 1 uschovaný kredit, který je využit na jeho obrácení v seznamu. Obě operace tak v případě amortizované analýzy dosahují konstantní složitosti. Pro ilustraci je na obrázku 2.4 vyobrazena asociace uschované jednotky kreditu s vloženým prvkem do seznamu *tail* a v tabulce 2.1 jsou shrnuty ceny operací.

Algorithm 2 Pseudokód operace odebrání prvku z funkcionální fronty.

```
Input:
    Q                                   ▷ The functional queue
 1: function DEQUEUE(Q)
 2:     if Q.head == NULL then
 3:         while Q.tail != NULL do
 4:             y ← Q.tail
 5:             Q.tail ← Q.tail.next
 6:             y.next ← Q.head
 7:             Q.head ← y
 8:         end while
 9:     end if
10:     x ← Q.head
11:     Q.head ← Q.head.next
12:     return x
13: end function
```



Obrázek 2.4: Funkcionální fronta se znázorněním uloženého kreditu (značeno R jako *resource*) u každého vloženého prvku [3].

Operace	Cena
ENQUEUE	$2 \cdot R$
DEQUEUE	R
REVERSE	R

Tabulka 2.1: Přehled cen jednotlivých operací u funkcionální fronty

Přehled vybraných složitostí. V [4] lze nalézt přehled asymptotických časových a prostorových složitostí některých často používaných datových struktur a algoritmů.

2.5 Existující řešení

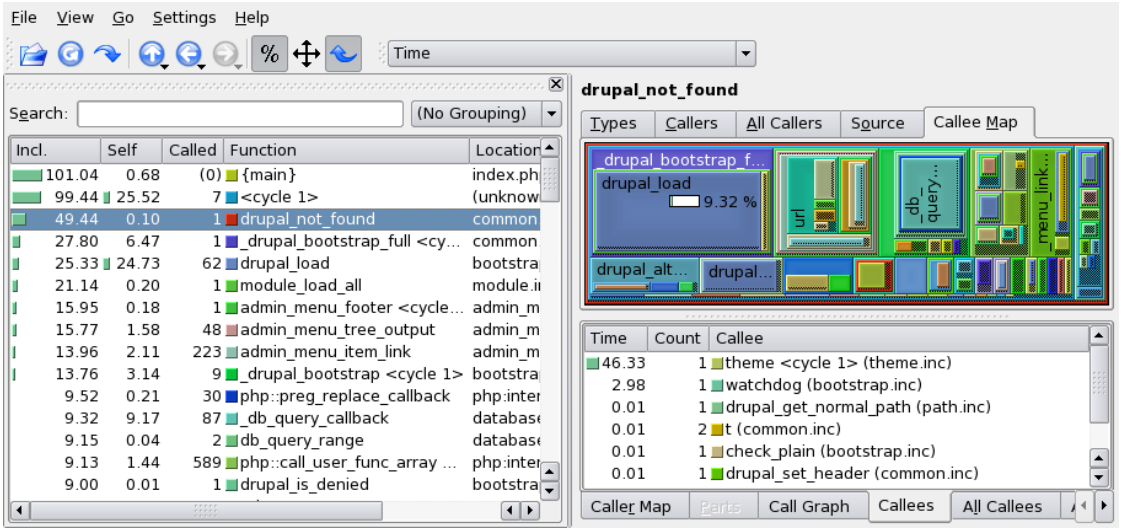
V současnosti existuje v oblasti profilace kvalitní sada nástrojů *Valgrind's Tool Suite* [22], která pokrývá oblasti profilace času, paměti, *cache* paměti a dalších výpočetních zdrojů. Z hlediska této práce je podstatná profilace času, jež je prováděna nástrojem *Callgrind*.

Callgrind sbírá data o volání všech funkcí v programu a uchovává je ve formě grafu volání. U každé funkce je zaznamenán počet provedených instrukcí a podíl na celkové době běhu programu. Tato analýza tak vývojáři poskytuje komplexní pohled na jednotlivé časové složky celkové doby běhu programu, a tím poskytuje klíčové informace k lokalizaci úzkých hrdel (tzv. *bottlenecks*) — kritických míst ve zdrojovém kódu, která se nejvíce podílejí na zpomalení běhu celého programu.

Jedná se o velmi robustní profilovací řešení produkující přesné výsledky, které jsou však získány za cenu výrazného prodloužení doby běhu profilovaného programu. Důvodem zpomalení je emulace běhu programu na syntetické CPU jednotce jádra Valgrindu [22]. Z tohoto důvodu Callgrind nemůže poskytovat výstupy v podobě reálných časových jednotek.

Callgrind neumožňuje detailní analýzu chování jednotlivých algoritmů vzhledem k velikosti jejich vstupu a není tedy schopen provést odhad jejich časových složitostí. Nástroj produkuje textový výstup, který je náročný na interpretaci uživatelem, avšak existuje externí grafická nadstavba v podobě *KCachegrind* nástroje [16]. Grafické rozhraní KCachegrindu je vyobrazeno ilustrací 2.5[3].

Práce si klade za cíl poskytnout nástroj, který adresuje nedostatky Callgrindu a umožní tak časově rychlejší profilaci, výstupy v reálných časových jednotkách, přehlednou vizualizaci jako součást nástroje a zaměření na časové složitosti algoritmů a operací nad datovými strukturami. Záměrem je tak poskytnout alternativu k současnému teoretickému způsobu odhadu časových složitostí popsanému v kapitole 2.



Obrázek 2.5: Ilustrace grafického rozhraní *KCachegrind* nástroje.

[3]Převzato z https://24b6.net/2009/06/11/kcachegrind-osx

Kapitola 3

Abstraktní datové struktury

Tato kapitola zavádí definici pojmu *abstraktního datového typu* a blíže popisuje vybrané abstraktní datové struktury [9]. Jsou popsány elementární datové struktury jako je fronta, zásobník nebo seznam a poté i komplexnější struktury jako binární vyhledávací strom nebo *skip list*. Některé z výše uvedených struktur jsou v závěru práce využity pro experimentální vyhodnocení vytvořeného nástroje.

3.1 Abstraktní datový typ

Abstraktní datový typ (*ADT*) je matematický model, který lze formálně definovat takto

Definice 3.1.1. *Abstraktní datový typ je definován množinou hodnot, jichž mohou nabývat prvky tohoto typu a množinou operací, definovaných nad tímto typem.[14]*

Jinými slovy je ADT teoretický koncept v informatice, který reprezentuje sadu prvků se společným chováním (sémantikou) a vlastnostmi nezávislými na konkrétní implementaci daného abstraktního datového typu[1]. Například množina celých čísel tvoří spolu s operacemi sjednocení, průnik a rozdíl příklad jednoduchého ADT [1].

3.2 Elementární datové struktury

Sekce se zběžně zabývá popisem elementárních datových struktur, které slouží jako základ struktur pokročilejších nebo jsou přímo využívány pro jejich implementaci. Konkrétně jsou představeny datové struktury zásobníku, fronty, jednosměrně vázaného seznamu, obousměrně vázaného seznamu a jejich varianty. Detaily k operacím, jejich implementaci a dodatečné podrobnosti k jednotlivým strukturám lze nalézt v [9].

3.2.1 Zásobník

Zásobník (*Stack*) je ADT reprezentující dynamickou množinu homogenních prvků stejného datového typu. Množina prvků je v zásobníku uspořádána pomocí systému *last-in, first-out* (*LIFO*), která definuje způsob vkládání a odebírání prvků dynamické množiny. Prvky zásobníku jsou tedy odebírány v pořadí opačném oproti jejich vložení. Vkládat prvky je možné pouze na *vrchol zásobníku* (tzv. *top*) a rovněž odebírání prvků lze provádět pouze z vrcholu. Vrchol zásobníku reprezentuje ukazatel na poslední vložený prvek v zásobníku.

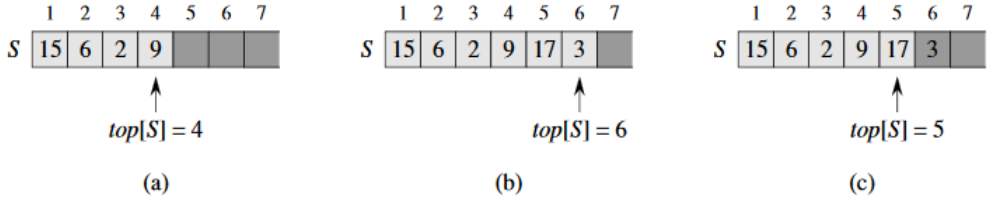
[1] https://en.wikipedia.org/wiki/Abstract_data_type

Operace. T. H. Cormen a spol. [9] definuje u zásobníku tři základní operace. Operace `POP(S)` vrací a zároveň odebírá prvek na vrcholu zásobníku S. `PUSH(S, x)` slouží k přidání prvku x do zásobníku S. Operace `STACK-EMPTY(S)` testuje prázdnost zásobníku S. Tabulka 3.1 ilustruje časovou složitost jednotlivých operací.

Operace	Časová složitost
`POP(S)`	$\mathcal{O}(1)$
`PUSH(S, x)`	$\mathcal{O}(1)$
`STACK-EMPTY(S)`	$\mathcal{O}(1)$

Tabulka 3.1: Přehled časových složitostí jednotlivých operací nad zásobníkem

Ilustrace. Obrázek 3.1 s vyobrazením zásobníku implementovaného pomocí pole.



Obrázek 3.1: Zásobník S s ukazatelem *top* na vrchol zásobníku (a), provedení operací `PUSH(S, 17)`, `PUSH(S, 3)` (b) a následně operace `POP(S)` (c) [9].

Aplikace. Zásobník nachází využití při vyhodnocování výrazů v postfixové notaci, algoritmech využívajících metodu zpětného navracení (*backtracking*) jako je třeba *depth-first search* (*DFS*) a rovněž další grafové algoritmy vyžadují pro svůj běh zásobník [2].

3.2.2 Fronta

Fronta (*Queue*) je stejně jako zásobník dynamickou množinou homogenních prvků. Liší se však systémem uspořádání, který je v případě fronty *first-in, first-out* (*FIFO*). Na rozdíl od zásobníku lze k frontě přistupovat ze dvou míst, z přední (tzv. *head*) a zadní (tzv. *tail*) části. Nově vložený prvek je umístěn do zadní části fronty a k odběru prvku dochází vždy z části přední. Oproti zásobníku jsou tedy prvky z fronty odebírány v pořadí, ve kterém do ní byly vloženy.

Varianty. Prioritní fronta (*Priority queue*) umožňuje přiřadit ke každému prvku hodnotu priority a prvky následně vkládat a vybírat podle jejich priorit. U oboustranné fronty (*Double-ended queue*) lze prvky přidávat nebo odebírat na obou koncích fronty. Existuje i jejich kombinace, tzv. prioritní oboustranná fronta (*Double-ended priority queue*).

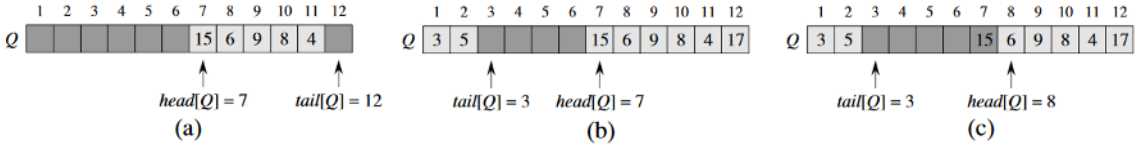
Operace. Podle [9] jsou nad frontou definovány dvě fundamentální operace, `ENQUEUE(Q, x)` a `DEQUEUE(Q)`. `ENQUEUE(Q, x)` způsobí vložení prvku x do zadní části fronty Q a operace `DEQUEUE(Q)` vrátí a zároveň odstraní prvek z její přední části. Tabulka 3.2 poskytuje podrobnosti k časové složitosti jednotlivých operací.

[2] https://en.wikipedia.org/wiki/Stack_(abstract_data_type)#Applications

Operace	Časová složitost
`ENQUEUE(Q, x)`	$\mathcal{O}(1)$
`DEQUEUE(Q)`	$\mathcal{O}(1)$

Tabulka 3.2: Přehled jednotlivých operací nad frontou a jejich časové složitosti.

Ilustrace. Obrázek 3.2 zobrazuje frontu implementovanou pomocí pole.



Obrázek 3.2: Fronta Q s ukazateli na přední a zadní část (a), po provedení operace `ENQUEUE(Q, 17)`, `ENQUEUE(Q, 3)`, `ENQUEUE(Q, 5)` (b) a následně po provedení operace `DEQUEUE(Q)` (c) [9].

Aplikace. Fronta je využívána u některých grafových algoritmů jako je například *Breadth-first Search* (*BFS*). Další uplatnění nachází v oblasti počítačových simulací pro implementaci kalendáře událostí (*Event Calendar*) nebo v oblasti synchronizace, kde mohou vznikat fronty z důvodu přístupu ke kritickým zdrojům. Fronty nachází taktéž uplatnění v oblasti operačních systémů nebo sítí, kde mohou zajišťovat zpracování požadavků ve správném pořadí.

3.2.3 Jednosměrně a obousměrně vázaný seznam

V datové struktuře seznamu (*list*) jsou prvky uspořádány lineárně za sebou. Jejich vzájemné provázání a uspořádání je určeno ukazateli v každém z prvků. Pokud prvek obsahuje pouze ukazatel na svého následníka (tzv. *next*), tak se jedná o jednosměrně vázaný seznam (*singly linked list*). Varianta, kdy prvek obsahuje ukazatel i na svého předchůdce (tzv. *prev*) se nazývá obousměrně vázaný seznam (*doubly linked list*). První prvek seznamu (neboli prvek, který nemá předchůdce) budeme označovat jako *head* a podobně poslední prvek (tentokrát ten, jenž nemá následníka) jako *tail*.

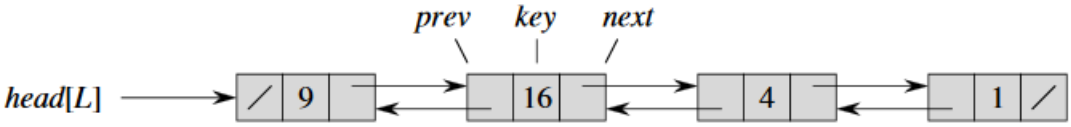
Varianty. U kruhového seznamu (*Circular list*) odkazuje první prvek kruhového seznamu na prvek poslední, jakožto na svého předchůdce. Podobně i poslední prvek označuje za svého následníka první prvek. Ve výsledku tak každý prvek obsahuje platný ukazatel na svého předchůdce i následníka. Seznam se zarážkou (*Sentinel node*) obsahuje na konci seznamu místo prázdného ukazatele (s hodnotou *NULL*) zarážku, což je zvláštní typ uzlu usnadňující operaci vyhledání prvku [2].

Operace. Podle T. H. Cormena a spol. [9] má datový typ seznamu definovány minimálně tři operace. Operace `LIST-INSERT(L, x)` a `LIST-DELETE(L, x)` umožňují vložit prvek x do seznamu L, respektive ho z něj odstranit. `LIST-SEARCH(L, k)` nalezne a vrátí prvek určený klíčem k v seznamu L. Tabulka 3.3 udává podrobnosti k časovým složitostem popsaných operací.

Operace	Časová složitost
`LIST-INSERT(L, x)`	$\mathcal{O}(1)$
`LIST-DELETE(L, x)`	$\mathcal{O}(1)$
`LIST-SEARCH(L, k)`	$\mathcal{O}(n)$

Tabulka 3.3: Přehled časových složitostí jednotlivých operací nad seznamem.

Ilustrace. Obrázek 3.3 vyobrazuje obousměrně vázaný seznam.



Obrázek 3.3: Obousměrně vázaný seznam s ukazatelem na první prvek *head* a popisem prvku seznamu [9].

Aplikace. Seznamy nachází uplatnění například v oblasti správy volné a obsazené paměti. *File Allocation Table* (*FAT*) využívá seznamy pro uchování informací o souborech na disku. Seznamy lze rovněž použít k efektivní implementaci zásobníku a fronty.

3.3 Binární vyhledávací strom

Datová struktura binárního vyhledávacího stromu (tzv. *Binary Search Tree*) je optimalizovaná pro operaci vyhledávání prvku. Každý prvek ve stromové struktuře obsahuje ukazatele na pravý (tzv. *right*) a levý (tzv. *left*) podstrom, klíč (tzv. *key*) a případně další data uchovávaná v prvku. Prvek stromu, jehož ukazatele na oba podstromy nabývají hodnoty *NULL*, je označován jako list (tzv. *leaf-node*).

Uspořádání klíčů ve struktuře musí být vždy takové, aby splňovalo vlastnost binárního vyhledávacího stromu. Nechť x je prvek v binárním vyhledávacím stromu, y je prvek v levém podstromu prvku x a z je prvek v pravém podstromu prvku x. Potom musí platit:

$$y.key \leq x.key \leq z.key \tag{3.1}$$

Časové nároky na provedení všech základních operací nad binárním vyhledávacím stromem jsou proporcionální vzhledem k výšce stromu. V zájmu zachování efektivnosti jednotlivých operací je tak vhodné mít strom vyvážený a jeho výška tedy musí odpovídat přibližně logaritmu počtu prvků n ve stromu ($\log n$). V nejhorším případě může nevyvážený binární vyhledávací strom degradovat až do stavu, kdy jeho výška odpovídá počtu prvků n. Operace tak nabývají lineární složitosti $\mathcal{O}(n)$ a struktura stromu pak připomíná seznam.

Varianty. AVL strom je tzv. samovyvažující se (*self-balancing*). Pokud přestane být strom po některé operaci vyvážený, tak dochází k jeho vyvážení pomocí rotací. Rovněž červenočerný strom (tzv. *Red-black tree*) patří do skupiny samovyvažujících se binárních vyhledávacích stromů. Tento typ stromu udržuje vyváženost stromu pomocí vlastnosti barvy u jednotlivých uzlů a podmínek, které tyto uzly musí splňovat vzhledem ke své barvě.

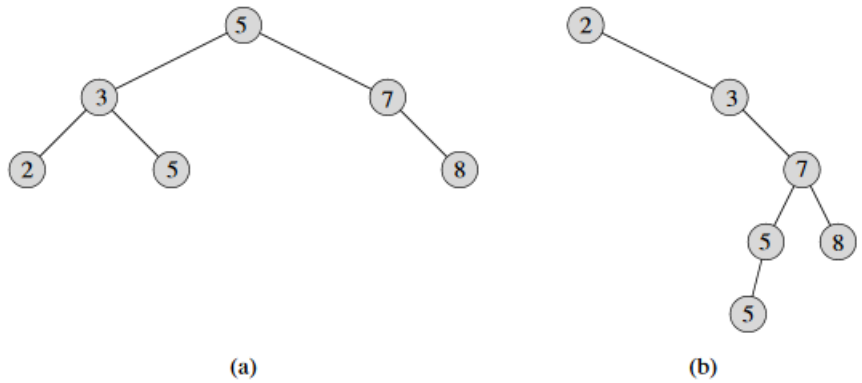
Při porušení některé z podmínek jsou určité uzly stromu přeuspořádány tak, aby byly podmínky splněny.

Operace. Binární vyhledávací strom obsahuje tři základní operace. `TREE-INSERT(T, z)` a `TREE-DELETE(T, z)` slouží ke vložení prvku z do stromu T, respektive k jeho odstranění ze stromu T. Operace `TREE-SEARCH(x, k)` vyhledá prvek podle klíče k postupným procházením stromu od jeho kořenového prvku x a vrátí ukazatel na nalezený prvek. V tabulce 3.4 jsou uvedeny časové složitosti jednotlivých operací.

Operace	Časová složitost	
`TREE-INSERT(T, z)`	$\mathcal{O}(n)$	$\Theta(\log n)$
`TREE-DELETE(T, z)`	$\mathcal{O}(n)$	$\Theta(\log n)$
`TREE-SEARCH(x, k)`	$\mathcal{O}(n)$	$\Theta(\log n)$

Tabulka 3.4: Přehled jednotlivých operací a jejich časových složitostí nad binárním vyhledávacím stromem.

Ilustrace. Obrázek 3.4 graficky ilustruje strukturu binárního vyhledávacího stromu.



Obrázek 3.4: Grafické znázornění struktury binárního vyhledávacího stromu s optimální výškou stromu (a) a jeho méně efektivní verze se stejnými, avšak jinak uspořádanými prvky (b) [9].

Aplikace. Binární vyhledávací strom nachází využití v případech, kdy je vyžadována efektivnost vyhledání prvku v datové struktuře, popřípadě průchod přes jednotlivé prvky (např. *in-order traversal*). Pomocí binárního vyhledávacího stromu lze rovněž vytvářet další abstraktní datové struktury, jako je například množina, multimnožina nebo asociativní pole[3].

[3] https://en.wikipedia.org/wiki/Binary_search_tree#Definition

3.4 Skip list

Skip list je pravděpodobnostní (*probabilistic*) datová struktura, která má podobné vlastnosti jako binární vyhledávací stromy. Na rozdíl od nich však skip list nemusí provádět vyvažování prvků (ať už explicitní, nebo jako součást operací v případě samovyvažujících struktur) a jedná se tak o rychlejší a snadnější alternativu stromových struktur [19]. Skip list udržuje své prvky implicitně vyvážené díky (pseudo)náhodě a udržování seřazenosti prvků. Žádná permutace vstupní sekvence nezpůsobí degeneraci struktury (oproti např. binárnímu vyhledávacímu stromu).

Skip list dosahuje zmíněných výhod díky modifikaci struktury jednosměrně (volitelně i obousměrně) vázaného seznamu. Každý prvek seznamu může obsahovat více než pouze jeden ukazatel — přesný počet určuje generátor (pseudo)náhodných čísel — čímž dochází k propojení nesousedících prvků. Tato modifikace tak prakticky vytváří více úrovní ukazatelů a tím umožňuje rychlejší průchod seznamem (a přeskočení nehledaných prvků).

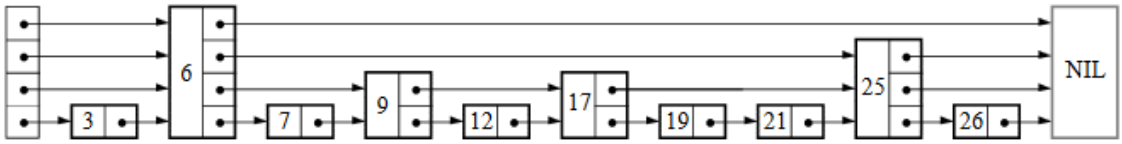
Operace. [19] řadí mezi základní operace skip listu vkládání, mazání a vyhledání prvků. `Insert(L, k, v)` vkládá hodnotu v s klíčem k do skip listu L. Operace `Delete(L, k)` způsobí odstranění prvku s klíčem k a `Search(L, k)` se pokusí vyhledat prvek specifikovaný klíčem k ve skip listu L. Tabulka 3.5 přehledně zobrazuje časové složitosti jednotlivých operací. Za povšimnutí rovněž stojí zmíněná paměťová složitost struktury.

Operace	Časová složitost	
`Insert(L, k, v)`	$\mathcal{O}(n)$	$\Theta(\log n)$
`Delete(L, k)`	$\mathcal{O}(n)$	$\Theta(\log n)$
`Search(L, k)`	$\mathcal{O}(n)$	$\Theta(\log n)$

Paměťová složitost
$\mathcal{O}(n \cdot \log n)$

Tabulka 3.5: Přehled jednotlivých operací a jejich časových složitostí nad skip listem. Paměťová složitost skip listu je zmíněna hlavně z důvodu její odlišnosti od ostatních struktur ($\mathcal{O}(n)$). Vysoká rychlost operací je tak zajištěna na úkor paměťových nároků.

Ilustrace. Obrázek 3.5 graficky ilustruje strukturu skip listu .



Obrázek 3.5: Grafické znázornění skip listu se čtyřmi úrovněmi ukazatelů. V případě hledání prvku se postupuje od nejvyšší úrovně směrem k úrovním nižším, dokud není prvek sekvenčně vyhledán v nejnižší úrovni [19].

Aplikace. Možným využitím skip listů je efektivní implementace vysoce škálovatelných konkurentních (*concurrent*) prioritních front nebo slovníků se sníženým počtem zámků (existují i tzv. *lockless* implementace nevyžadující vůbec žádné zámky). Dále lze skip list využít pro zefektivnění některých statistických výpočetních operací. Z praktických aplikací lze zmínit databázový systém *MemSQL*, který využívá skip listy jako svou hlavní indexovací datovou strukturu[4].

[4] https://en.wikipedia.org/wiki/Skip_list#Usages

Kapitola 4

Nástroj Perun

Perun (*Performance Under Control*) je nástroj vyvíjený vedoucím práce Ing. Fiedorem [12], jehož cílem je umožnit snadný a automatizovatelný přístup k profilaci programů během jejich vývoje. Hlavní inspirací projektu jsou systémy pro správu verzí (tzv. *version control systems*), které uchovávají informace o změnách v projektu (např. *Git* nebo *SVN*). Nástroj Perun má za cíl automaticky vytvářet a spravovat výkonnostní analýzy k příslušným stavům projektu tak, aby informace o výkonnostní stránce byly rovněž součástí historie vývoje projektu bez nutnosti manuálního zapojení uživatele do celého procesu.

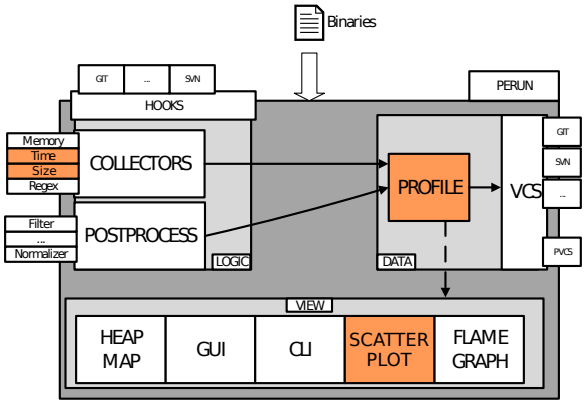
Autor této práce se podílel na návrhu architektury nástroje a jeho komunikačního rozhraní, přičemž výsledná knihovna je v tomto nástroji integrována (integrace některých celků ale ještě není plně realizována). Některé části práce jsou na nástroji závislé, a proto se tato kapitola věnuje stručnému představení architektury a především využitého rozhraní Perunu.

4.1 Architektura

Na nejvyšší úrovni lze Perun rozdělit do tří částí — dat, pohledu a logiky. Popisovaná architektura je ilustrována obrázkem 4.1.

Data. V datové části se nachází správa *profilu*, což je základní datová jednotka, se kterou nástroj pracuje. Tento datový formát má z hlediska obecnosti a rozšiřitelnosti nástroje velký význam, a proto se mu podrobněji věnuje sekce 4.2. Kromě profilu se v sekci dat nachází také výstupní napojení na podporované verzovací systémy.

Logika. Tato část zahrnuje vytváření a správu jednotlivých profilů. O vytváření profilů se starají jednotky zvané *kolektory*, jejichž výstupy lze dále zpracovávat pomocí tzv. *postprocess* modulů. Kapitola 5 detailně představuje realizaci kolektoru časových dat a dat o velikosti struktury,



Obrázek 4.1: Architektura nástroje Perun ilustrující jeho logickou, datovou a vizualizační část včetně příslušných modulů. Šipky značí vzájemnou komunikaci mezi jednotlivými moduly a barevně je vyznačen přínos autora této práce [12].

který je součástí této práce a byl integrován do nástroje Perun. Kromě zmíněných modulů se v části nachází rovněž napojení (tzv. *hooks*) na verzovací systémy. Tato napojení realizují užší integraci Perunu do vývojového procesu daného verzovacího systému (např. v rámci systému git umožňuje automaticky vytvořit sadu profilů během každého *commitu* změn.)

Pohled. Jedná se o částečně nezávislý modul, který zajišťuje vstupní a výstupní rozhraní pro interakci s uživatelem. Součástí modulu je grafické a textové uživatelské rozhraní, které je doplněno o výstupní vizualizace analyzovaných profilovacích dat.

4.2 Životní cyklus profilu

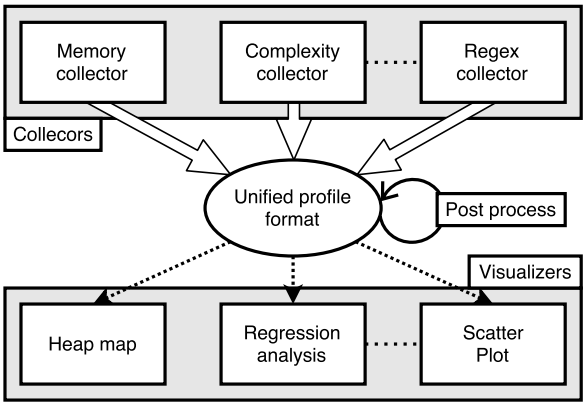
Pro uchování profilovacích informací byl všemi spoluautory Perunu navržen uniformní datový formát inspirovaný formátem *JSON* [15], který slouží jako jednoduché přenositelné rozhraní mezi veškerými moduly, jež s profilem manipulují. Díky jednotnému formátu je možné snadno rozšiřovat stávající sestavu kolektorů, analyzátorů a vizualizérů. Ukázka formátu profilovacího souboru je na obrázku 4.2

Jádrem automatizace v Perunu jsou tzv. matice práce (*job matrix*) určené množinou kolektorů, postprocesorů a vstupních dat. Pro každá vstupní data (binární soubory, zdrojové kódy atd.) jsou jednotlivě spuštěny kolektory generující profily, které jsou následně zpracovány pomocí série postprocesorů. Každá matice tedy vygeneruje sadu profilů, které mohou být uloženy do systému Perun. Společně s profily lze ukládat i dodatečné informace jako např. verzi programu, ke které se profil váže. Generované profily je pak možno vizualizovat pomocí různých vizualizačních technik. Celý proces je podrobně ilustrován obrázkem 4.3.

```
Profile = {
 'header': {
  'type': 'mixed',
  ...
 }

 'global': {
  "resources": [
   {
    "amount": 61,
    "structure-unit-size": 0,
    "subtype": "time delta",
    "type": "mixed",
    "uid": "skiplistInsert(skiplist*, int)"
   },
   {
    "amount": 13,
    "structure-unit-size": 1,
    "subtype": "time delta",
    "type": "mixed",
    "uid": "skiplistSearch(skiplist*, int)"
   },
   ...
```

Obrázek 4.2: Ukázka jednotného formátu pro uložení profilovacích dat.



Obrázek 4.3: Grafická ilustrace životního cyklu profilace včetně znázornění role samotného profilu jakožto rozhraní modulů.

Kapitola 5

Sběr profilovacích dat

Tato kapitola se zabývá návrhem a realizací sběru profilovacích dat, která budou dále použita k aproximaci časových složitostí algoritmů, jako jsou například operace nad datovými strukturami. Sběr dat je realizován pomocí logické jednotky zvané *kolektor*, která je založena na principu odlehčené instrumentace profilovaného programu. V první části kapitoly je provedena analýza samotného problému sběru dat a jsou vytyčeny cílové požadavky na implementaci kolektoru. Následně je představen navrhovaný způsob sběru dat a závěr kapitoly se věnuje praktické implementaci navrženého kolektoru.

5.1 Analýza

Na teoretické úrovni je odhad časové složitosti algoritmu prováděn analýzou zdrojového kódu, kdy je doba běhu charakterizována jako funkce velikosti vstupních dat. Čas je reprezentován počtem provedených primitivních operací a velikost vstupních dat je vyjádřena formou konstanty n (viz kapitola 2). Pro praktický odhad složitosti je však potřeba nahradit tyto abstraktní parametry skutečně naměřenými daty. Výsledný kolektor tedy musí umožňovat sběr jak *časových dat*, tak *dat o velikosti vstupu*.

Časová data Časová data reprezentují reálné množství času, které algoritmus spotřeboval nad daným vstupem. V ideálním případě by měla být časová data získávána z každého volání daného algoritmu, což by vedlo k dosažení vysoké přesnosti samotného odhadu složitosti. U některých algoritmů však může tento přístup zapříčinit nepřijatelné výpočetní a paměťové nároky, a proto musí být implementováno vzorkování sbíraných dat za běhu kolektoru.

Data o velikosti vstupu Samotná časová data ale neposkytují dostatek informací pro odhad složitosti, a tak je ke každému časovému záznamu potřeba získat rovněž informaci o množství dat, nad kterými algoritmus pracoval. Naměřená data o velikosti vstupu vztažená ke konkrétnímu časovému záznamu tak tvoří dvojici $D = (T, S)$, kde T reprezentuje dobu běhu algoritmu a S množství dat, nad kterými algoritmus pracoval. Množina $C = \{D_0, D_1, \ldots, D_n\}$ dvojic D_x reflektuje závislost doby běhu na velikosti dat a umožňuje následně odvodit konkrétní model složitosti (více viz kapitola 6). Cílem kolektoru je naměření této množiny pro daný program.

5.1.1 Požadavky

Na výsledný datový kolektor je kladena řada nároků a požadavků, které musí být při návrhu a implementaci brány v potaz. Ne všechny požadavky je však možné bez výhrad splnit, především z důvodu kompromisů mezi rychlostí, přesností a paměťovými nároky kolektoru.

1. **Malá režie.** Kolektor by měl co nejefektivněji instrumentovat sledovaný program tak, aby nedocházelo k výraznému prodloužení doby běhu samotného programu způsobeného vysokou mírou režie sběru dat. Cílem je vyhnout se situaci při použití nástrojů z *Valgrind's Tool Suite* [22], jejichž dokumentace uvádí možné zpomalení běhu programu 10–30 krát (nástroj *Memcheck*) nebo až 20–100 krát v případě nástroje *Cachegrind*. Cenou za nižší režii kolektoru pak může být případné snížení přesnosti dat.

2. **Přesnost dat.** Prováděním své vnitřní logiky by kolektor měl minimalizovat zkreslení měřených dat, konkrétně měřenou dobu běhu algoritmu.

3. **Minimum závislostí.** Cílem je minimalizovat závislosti kolektoru na knihovnách třetích stran, konkrétních překladačích a specifických systémových nástrojích.

4. **Minimum manuálních zásahů v kódu.** Kolektor by měl vyžadovat minimum úprav profilovaného kódu ve smyslu přidávání anotací, dodatečných proměnných nebo profilovacích modulů a hlavičkových souborů do zdrojového kódu. Celkově tedy minimalizovat manuální zapojení uživatele do procesu sběru dat.

5.2 Návrh

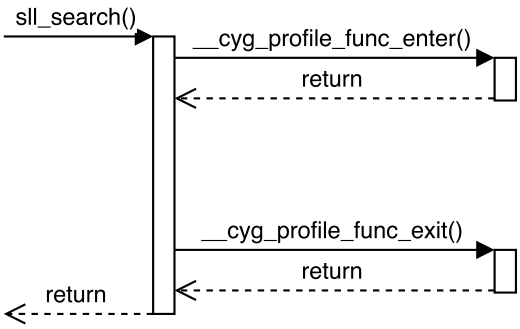
Tato sekce se zaměřuje na popis navrhovaného řešení pro sběr časových dat a dat o velikosti vstupu. U návrhu jsou diskutovány jeho výhody a nevýhody spolu s některými problémy, které se v průběhu návrhu a implementace objevily a které musí kolektor spolehlivě řešit.

5.2.1 Časová data

Algoritmy nebo operace nad datovými strukturami jsou v jazyce C/C++ standardně implementovány v podobě funkcí nebo metod. Čas potřebný pro vykonání těchto funkcí tedy odpovídá časovým údajům, které mají být kolektorem shromažďovány. *GCC* (*GNU Compiler Collection*) umožňuje instrumentovat funkce za pomoci speciálních vestavěných funkcí, jejichž volání je při překladu vkládáno (tzv. *code injection*) do vstupního a výstupního bodu funkce (konkrétně na začátek a konec těla funkce), jak ilustruje obrázek 5.1.

Těla těchto instrumentačních funkcí — a tedy vnitřní logiku instrumentace — lze implementovat libovolným způsobem. Z pohledu časových dat je vhodné v obou instrumentačních funkcích generovat časová razítka (tzv. *timestamp*), jejichž rozdíl pak odpovídá době běhu funkce. Dále je potřeba dosáhnout dostatečné přesnosti časových razítek, aby bylo možné spolehlivě měřit dobu běhu i na rychlejších procesorech, případně u velmi rychlých funkcí. Knihovna `chrono` [6], která je součástí standardu `C++11`, umožňuje zachycení časového údaje až v řádu nanosekund, a proto byla zvolena pro implementaci časových razítek.

Výhody. GCC instrumentace pracuje na odlehčeném principu a zároveň poskytuje vysokou míru kontroly nad samotnou vnitřní logikou, která je prováděna v těle instrumentačních funkcí. Vkládání kódu je prováděno v době překladu a je plně v kompetenci překladače, což minimalizuje režii za běhu programu a umožňuje například instrumentovat i *inline* funkce.



Obrázek 5.1: Grafická ilustrace principu instrumentace funkcí pomocí GCC přepínače `-finstrument-functions` [13]. Volání funkcí `__cyg_profile_func_enter` a `__cyg_profile_func_exit` je prováděno bezprostředně po vstupu do těla funkce `sll_search`, respektive před výstupem z něj. Logiku těchto instrumentačních funkcí je ale nutné implementovat ve vlastní režii.

Nevýhody. Přestože řešení nabízí kontrolu nad instrumentační logikou, neumožňuje modifikovat parametry volaných funkcí. Tento způsob instrumentace rovněž vytváří závislost na konkrétní rodině překladačů GCC, nicméně v současnosti nabízí podobné řešení i překladač *Clang* [7]. Rozšiřování kolektoru dat o další podporované platformy a překladače (případně sběr dat na nižší úrovni) je jedním z možných cílů navazující práce.

Problémy. Zpravidla není nutné provádět instrumentaci všech funkcí uvnitř programu, ale pouze těch, které jsou z pohledu profilování důležité nebo zajímavé. Instrumentace veškerých funkcí přináší riziko příliš vysokých výkonnostních nebo paměťových nároků a zároveň zbytečně detailního výstupu. Stejný problém může nastat i v případě, kdy je filtrování funkcí podporováno, ale instrumentace přesto poskytuje enormní množství profilovacích dat. Kolektor proto musí implementovat spolehlivé a odlehčené řešení filtrace a vzorkování profilovaných funkcí. Sekce 5.2.3 se zabývá podrobnějším popisem problematiky a navazující sekce 5.2.4 popisem navrženého řešení.

V současnosti není v kolektoru vyřešena problematika rekurzivního zanořování funkcí. Aktuálně není rozhodnuto o způsobu, jakým by měla být měřena doba běhu takto vnořených funkcí (mezi možné varianty řešení patří např. měření *inkluzivní* nebo *exkluzivní* doby běhu) a jak řešit problémy plynoucí z nepřímé rekurze funkčních volání.

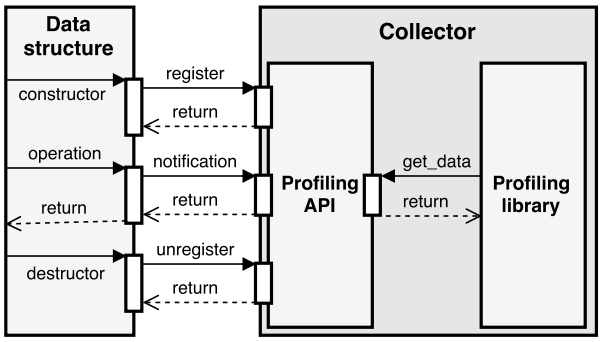
5.2.2 Data o velikosti vstupu

Množství dat na vstupu algoritmu — na rozdíl od časových dat — nelze vždy snadno odvodit nebo změřit. V závislosti na implementaci získávají algoritmy tento údaj jako vstupní parametr, jiné jako atribut třídy nebo struktury, nad kterou pracují, případně jej zjišťují voláním příslušných funkcí a metod. Stejně tak některé datové struktury nemusí tuto informaci vůbec potřebovat, případně není nikde uložena a dochází k jejímu výpočtu za běhu algoritmu. Sjednocení lze docílit zavedením uživatelských anotací, které zajistí korektní sběr dat napříč veškerými scénáři za cenu nutného zásahu uživatele.

Bylo navrženo profilovací *API* [1], které má zajistit podobnou funkcionalitu jako anotace zdrojového kódu, avšak s přidanou hodnotou v podobě obecnějšího použití a snadné rozšiřitelnosti. Rozhraní pracuje na principu registrace sledovaných struktur (např. vstupního pole řadicího algoritmu nebo datové struktury s jednosměrně vázaným seznamem) spolu s jejich velikostí a následné notifikace v případě práce s danou strukturou (provádění algoritmu nad registrovanou strukturou). Při návrhu byl kladen důraz na snadnou a komfortní práci s rozhraním, které proto poskytuje i možnost vestavět volání funkcí z rozhraní do kódu profilovaného algoritmu nebo operace datové struktury. Díky tomu je možné provádět a rušit registrace v konstruktorech a destruktorech, což umožňuje využití *RAII (Resource acquisition is initialization)* principu. Práce s profilovacím API je ilustrována obrázkem 5.2.

[1] *Application Programming Interface*

Výhody. Profilovací rozhraní je navrženo s ohledem na snadnou rozšiřitelnost (např. o klasickou analýzu amortizované složitosti nebo sledování spotřeby jiných systémových zdrojů jako je práce se soubory), a díky tomu je do budoucna využitelné v širším spektru profilovacích nástrojů. Operace nad profilovacím rozhraním lze provádět uvnitř samotného zdrojového kódu algoritmu ale i mimo něj, což přináší větší flexibilitu a umožňuje profilovat např. *STL* [2] struktury, u kterých nelze přímo modifikovat zdrojový kód. Stejně tak je možné profilovat i datové struktury bez informací o své velikosti, což ale vyžaduje více vlastní režie z pohledu uživatele.



Obrázek 5.2: Demonstrace principu profilovacího API, jehož volání bylo vestavěno do kódu operací nad strukturou. Při vytvoření struktury konstruktorem dochází k její registraci (`register`) a při destrukci ke zrušení registrace (`unregister`). U profilovaných operací je zaslána notifikace (`notification`) interním profilovacím strukturám, jejichž data mohou být následně zpřístupněna profilovací knihovně (`get_data`).

Nevýhody. Navržené rozhraní lze chápat jako jistou formu anotace zdrojového kódu a z toho plynoucí problémy s nutností modifikovat profilovaný kód — samotný návrh však provázela snaha o minimalizaci nezbytně nutných úprav profilovaného kódu. Současný návrh profilovacího API neumožňuje profilovat funkce, jejichž zdrojový kód ani samotné volání není přístupné uživateli (např. interní instance struktur využívané ve standardních knihovnách)

Problémy. Stejně jako v případě časových dat není v současnosti podporována rekurze funkčních volání. Přestože navržené rozhraní dokáže rekurzi korektně zpracovat, tak tato data nelze párovat s daty časovými.

5.2.3 Konfigurace kolektoru

Sekce 5.2.1 naznačila důvody vedoucí k nutnosti podporovat v rámci kolektoru filtrování a vzorkování instrumentovaných funkcí — konkrétně příliš detailní výstupy nebo nadměrné výpočetní a paměťové nároky. GCC pro tyto požadavky neposkytuje dostatečně robustní řešení, které tak musí být efektivně implementováno v režii kolektoru.

Filtrování funkcí. Spolu s přepínačem pro instrumentaci funkcí nabízí GCC i přepínač `-finstrument-functions-exclude-function-list` [13] umožňující specifikovat seznam identifikátorů funkcí, které nebudou instrumentovány. Toto řešení však není z pohledu uživatele příliš praktické, neboť vyžaduje znalost veškerých funkcí ve výsledném binárním souboru a vytvoření doplňku k požadovaným funkcím — to vše před samotným sestavením programu. Seznam výjimek je navíc porovnáván i na shodu v podřetězci, kdy je filtrována funkce s identifikátorem obsahujícím podřetězec identifikátoru uvedeného v seznamu výjimek, a tak může dojít k filtraci nespecifikovaných funkcí. Výhodou seznamu výjimek je

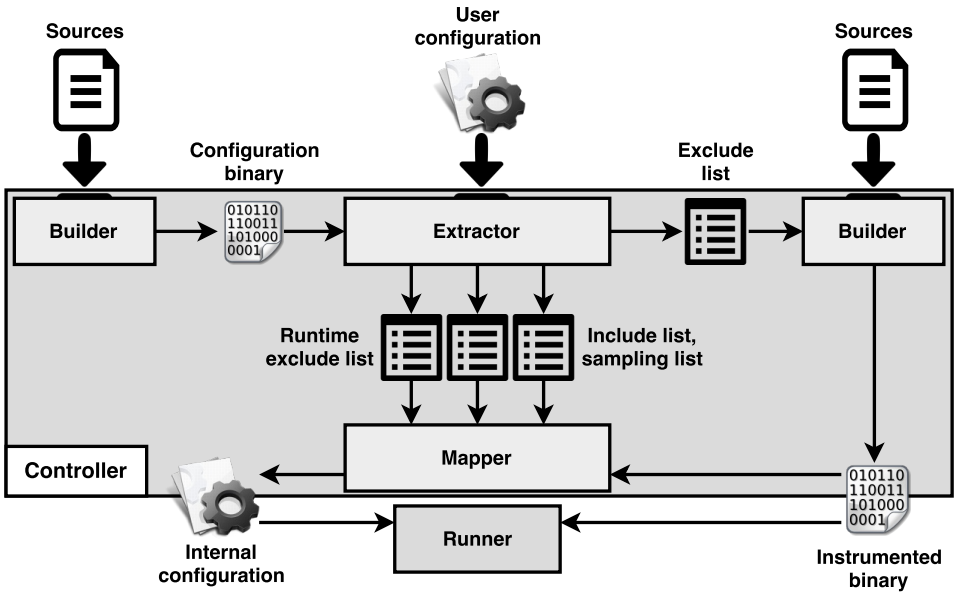
[2] *Standard Template Library*

filtrace už v době překladu (filtrované funkce tak nejsou vůbec instrumentovány), což má za následek snížení režie kolektoru. Vhodným řešením je tedy využití seznamu výjimek, pokud je to možné, a pro zbytek případů aplikovat filtraci za běhu programu.

Vzorkování. Vzorkování instrumentace lze provádět až za běhu programu a z tohoto důvodu nemůže GCC poskytovat žádnou podporu. Veškeré operace spojené se vzorkováním jsou tak v režii kolektoru. Za běhu programu musí kolektor nakonfigurovat interní profilovací logiku tak, aby korektně docházelo k zahazování nežádoucích instrumentačních dat.

5.2.4 Navrhované řešení

Kolektor má na svém vstupu uživatelskou konfiguraci (obsahující uživatelské nastavení kolektoru jako např. seznam profilovaných funkcí, specifikaci vzorkování jednotlivých funkcí atd.) a cesty ke zdrojovým souborům, které instrumentuje a překládá do výsledného binárního kódu. Sestavení profilovaného binárního souboru předchází mezikrok — vytvoření konfiguračního binárního souboru. Jedná se o binární soubor přeložený ze zdrojových kódů bez vložené instrumentace a sloužící k extrakci tabulky funkčních symbolů. S pomocí tabulky symbolů a vstupní konfigurace je vytvořen seznam výjimek aplikovatelný při překladu, seznam výjimek filtrovaných za běhu, seznam profilovaných funkcí a specifikace vzorkování.



Obrázek 5.3: Schematický návrh kolektoru, který v první fázi transformuje vstupy uživatele na výsledný instrumentovaný binární soubor a interní konfigurační soubor kolektoru. Druhou fází kolektoru je aplikace konfiguračního souboru na profilovací logiku a samotné spuštění instrumentovaného binárního souboru.

Po dokončení extrakce je provedeno sestavení výsledného binárního souboru (překlad je tedy prováděn dvakrát) s instrumentací a aplikovaným seznamem výjimek. Nad výsledným souborem je znovu provedena extrakce tabulky symbolů. Extrakce slouží k namapování adres funkcí na jejich prototypy tak, aby bylo možné identifikovat aktuálně instrumentovanou funkci (adresa funkce je jeden z parametrů instrumentačních funkcí GCC) a dle konfigurace ji filtrovat nebo vzorkovat. Tato interní konfigurace (tzv. *ComplexityCollector Internal*

Runtime Configuration (CIRC) [3]) je za běhu programu kolektorem zpracována a aplikována na profilovací logiku. Jakmile je kolektor nakonfigurován, začíná samotný proces sběru dat spuštěním binárního souboru s instrumentací. Celý postup je ilustrován schématem 5.3.

5.3 Implementace

V této sekci je stručně představena implementace kolektoru dat a jeho dvou nejdůležitějších částí — tzv. *Runner* a *Controller* částí. Jako `Controller` je označována část kolektoru zodpovědná za transformaci vstupních dat na instrumentovaný binární soubor a interní konfiguraci kolektoru. `Runner` odkazuje na stejnojmenný prvek ze schématu 5.3, jehož úkolem je provést samotný sběr a uložení profilovacích dat z instrumentovaného binárního souboru s využitím dodané interní konfigurace.

5.3.1 `Runner`

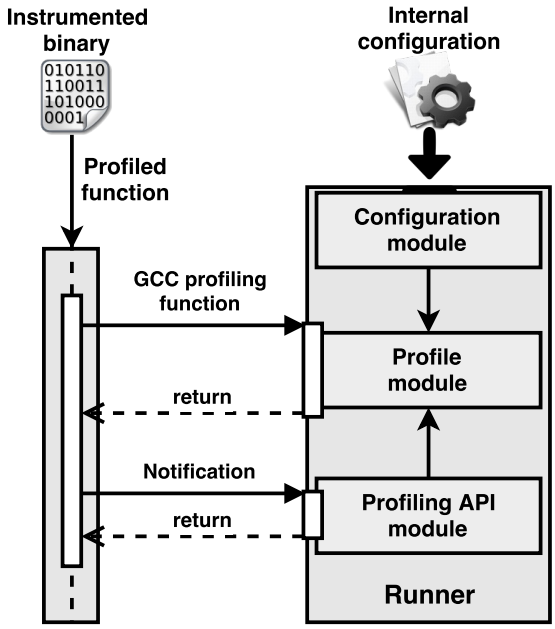
Na tuto část kolektoru jsou kladeny vysoké nároky v podobě nízké režie, vysoké rychlosti a paměťové nenáročnosti, aby nedocházelo k přílišnému zpomalení vlivem sběru dat. Tyto nároky — spolu se skutečností, že `Runner` pracuje s instrumentačními funkcemi GCC (viz 5.2.1) a profilovacím API (sekce 5.2.2) — vedly k výběru C++ jakožto vhodného implementačního jazyka.

Modul `configuration`. Jedná se o modul zajišťující zpracování a analýzu interního konfiguračního souboru (popsaného v sekci 5.2.4) pomocí třídy `Configuration`. Získaná data (např. adresy filtrovaných funkcí, adresy vzorkovaných funkcí spolu s koeficientem vzorkování a další informace převážně o vnitřní konfiguraci modulu) jsou uložena do datové struktury `func_config` umožňující rychlé vyhledání adresy a k ní příslušným konfiguračním detailům.

Modul `profile_api`. Smyslem modulu je poskytovat rozhraní (v podobě hlavičkového C/C++ souboru) pro sběr dat o velikosti vstupu algoritmů (sekce 5.2.2). Modul obsahuje interní datové struktury `struct_size_map` pro uložení informací o registrovaných strukturách a `size_stack` sloužící jako zásobník přijatých notifikací. Profilovací rozhraní zpřístupňuje uživateli následující sadu funkcí:

- `_profapi_register_size_address / _profapi_register_size_value:` provádí registraci struktury a k ní příslušné adresy nebo hodnoty s velikostí struktury.
- `_profapi_unregister_size:` slouží ke zrušení registrace struktury.
- `_profapi_using_size_adress / _profapi_using_size_value:` posílá notifikaci o použití struktury.
- `_profapi_get_size_record:` umožňuje nalézt záznam o aktuálně známé velikosti struktury.

[3] *Náhled dostupný na Github repozitáři* https://github.com/JiriPavela/perun/blob/complexity-collector/docs/complexity/circ.rst



Obrázek 5.4: Demonstrace spolupráce jednotlivých modulů v prvku *Runner*. Modul `configuration` poskytuje detaily o konfiguraci instrumentační logiky modulu `profile`, který získává instrumentační data z profilovacích GCC funkcí a modulu `profile_api`.

Modul `profile`. Modul využívá oba předchozí moduly a implementuje samotnou instrumentační logiku GCC funkcí — tedy filtrování, vzorkování a ukládání profilovacích údajů. Inicializace modulu (mimo jiné i konfigurace `Runneru` za pomocí `configuration` modulu) je provedena před samotným spuštěním profilovaného programu. Data jsou ukládána do interního bufferu `instr_data` a výstupního profilovacího souboru. Obrázek 5.4 ilustruje propojení jednotlivých modulů a jejich vzájemnou interakci.

Implementační výstup. Výstupem implementace jsou dvě sdílené knihovny (tzv. *dynamic library*) `libprofile.so` a `libprofapi.so`. První z knihoven obsahuje moduly `configuration` a `profile`, zatímco druhá pouze modul `profile_api`. Sběr dat je proveden spuštěním instrumentovaného binárního souboru, který byl spojen (*přilinkován*) s oběma knihovnami. Při profilaci jsou data ukládána do výstupního souboru specifikovaného ve vstupní konfiguraci kolektoru.

Rozdělení do dvou samostatných knihoven je praktické zejména z hlediska rozšiřitelnosti a znovupoužitelnosti knihovny profilovacího API pro širší spektrum nástrojů. Tyto nástroje však musí být s knihovnou kompatibilní z hlediska *ABI*[4].

5.3.2 `Controller`

Na rozdíl od `Runner` části nemá `Controller` tak striktní požadavky na výkon a efektivitu (nižší efektivita zde nemůže např. způsobit zkreslení dat), nepracuje s vestavěnými funkcemi GCC ani profilovacím API — naopak je o úroveň výš než `Runner`, zastřešuje jeho funkcionalitu, využívá systémová volání a nástroje operačního systému. Zároveň se jedná o prvek implementující rozhraní nástroje *Perun* pro kolektory dat. Nástroj a jeho rozhraní jsou implementovány v jazyce *Python*, a proto byl tento jazyk vybrán jako implementační. `Controller` sestává z modulů `makefiles`, `symbols`, `configurator` a `complexity`.

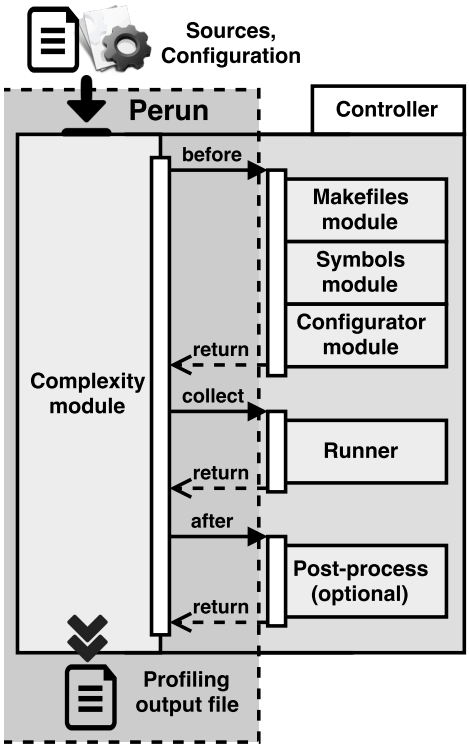
Modul `makefiles`. Modul `makefiles` implementuje rozhraní pro přeložení a sestavení binárních souborů ze zdrojového kódu — plní tedy úlohu prvku *Builder* ze schématu 5.3. Proces překladu je prováděn ve dvou krocích. Ze zadaných vstupů je nejprve vygenerován soubor *CMakeLists*, který je v dalším kroku využitý nástrojem *CMake* [8] pro samotný překlad.

[4] *Application Binary Interface*

Modul `symbols`. Modul obstarává úlohy prvků *Extractor* a *Mapper* ze schématu 5.3. S využitím nástroje *readelf* [20] je z binárního souboru extrahována tabulka funkčních symbolů, která je následně v režii modulu zpracována a analyzována. Cílem analýzy symbolů je jejich rozdělení do příslušných seznamů (seznam výjimek při překladu, seznam výjimek za běhu atd.). Je proveden rozklad funkčních symbolů na několik částí, které usnadňují samotný proces analýzy a umožňují zpracovat širší spektrum formátů vstupní konfigurace.

Modul dále poskytuje rozhraní pro vytvoření potřebných map symbolů, mezi které patří například mapování adres funkcí na jejich prototypy nebo mapování dekorovaných (tzv. *mangled*) jmen na nedekorovaná (tzv. *demangled*)[5].

Modul `configurator`. Modul `configurator` slouží ke generování interních konfiguračních souborů využívaných částí `Runner`. Vstupy modulu jsou seznamy funkčních symbolů a jejich mapování poskytované modulem `symbols`. Vzhledem ke své ryze implementační povaze nemá tento modul přímý ekvivalent některého z prvků na schématu 5.3.



Obrázek 5.5: Schéma jednotlivých modulů a rozhraní v `Controller` části kolektoru.

Modul `complexity`. Jedná se o modul zastřešující a koordinující práci tří ostatních modulů. Implementuje rozhraní vynucené nástrojem Perun, pomocí kterého je Perun schopen řídit proces sběru dat. Výstupem modulu je unifikovaný profilovací soubor popsaný v sekci 4.2. Rozhraní se skládá z následujících funkcí:

- `before` zajišťuje provedení všech přípravných akcí před samotným sběrem dat — tedy přípravu instrumentovaného binárního souboru a interní konfigurace ze vstupních souborů. Obsahuje většinu logiky *Controller* části.
- `collect` provádí samotný sběr dat voláním prvku `Runner` a kontrolu korektního dokončení této fáze. Výstupem je interní profilovací soubor obsahující časová razítka a korespondující velikost struktury pro všechna volání profilovaných funkcí.
- `after` je volitelná součást rozhraní, která transformuje profilovací soubor vyprodukovaný `collect` fází na unifikovaný datový formát.

Obrázek 5.5 graficky ilustruje využití a spolupráci jednotlivých modulů při sběru dat. Schéma rovněž zobrazuje rozhraní Perunu, které je v `complexity` modulu implementováno a pomocí kterého je sběr dat řízen.

[5] Jedná se o způsob jednoznačné identifikace symbolů v rámci překladače u jazyků, které principiálně nemohou jednoznačně identifikovat symbol pouze podle jeho jména (např. v případě podpory přetěžování funkcí).

Kapitola 6

Regresní analýza

Regresní analýza je oblast statistiky, která se zabývá hledáním vztahu mezi dvěma nebo více proměnnými. Konkrétně se jedná o vztah hodnot *závislých proměnných* na hodnotách *nezávislých proměnných*, kde výslednou závislost vyjadřujeme formou *regresní funkce*.

Regresní analýza je členěna do několika skupin, přičemž z pohledu práce je nejdůležitější rozdělení na lineární a nelineární regresní analýzu. Lineární regresní analýza pracuje se specifikací modelu, jehož *závislá proměnná* je lineární kombinací parametrů. Z těchto důvodů jsou do lineární regrese řazeny modely lineární, kvadratické nebo kubické (obecně polynomiální) a do nelineární regrese například modely mocninné, logaritmické nebo exponenciální. Nelineární regresní modely nemají snadné řešení, avšak některé lze transformovat na modely lineární (jsou tzv. *intrisically linear*), které lze řešit zavedenými postupy. Teoretický základ této kapitoly vychází převážně z [10].

6.1 Analýza

Pro odhad časové složitosti operací nad datovými strukturami (obecně algoritmů, jejichž doba běhu je ovlivněna počtem vstupních dat) je využita oblast regresní analýzy zvaná jednoduchá lineární regrese (tzv. *simple linear regression*). Výpočty jsou prováděny pouze nad dvěma proměnnými — konkrétně nad *nezávislou* proměnnou v podobě velikosti datové struktury a *závislou* proměnnou reprezentovanou dobou běhu algoritmu. Požadované nelineární modely (logaritmické, mocninné a exponenciální) byly transformovány na modely lineární. Cílem je nalézt model, který nejlépe vystihuje závislost času na velikosti a umožní tak provést odhad doby běhu algoritmu nad různým množstvím dat.

Problém je řešitelný pomocí *metody nejmenších čtverců*. Princip metody spočívá v nalezení parametrů modelové funkce $f(x, \beta)$ (β značí vektor parametrů) takových, aby výsledná suma čtverců chyb R_i 6.1 nabývala minima. Zmíněná chyba R_i reprezentuje rozdíl 6.3 naměřené hodnoty y_i a hodnoty regresního modelu $\widehat{y_i}$ v daném bodě 6.2.

$$S = \sum_{i=1}^{n} R_i^2 \tag{6.1}$$

$$\widehat{y_i} = f(x_i, \beta) \tag{6.2}$$

$$R_i = y_i - \widehat{y_i} \tag{6.3}$$

Řešení této optimalizační úlohy se liší v závislosti na regresním modelu (pro různé modely jsou mírně odlišné výpočetní rovnice). Za účelem sjednocení (a snadné rozšiřitelnosti) byl hledán univerzální vzorec, pomocí kterého by bylo možné provést výpočet libovolného modelu — i za cenu snížené přesnosti. Výsledné univerzální vzorce 6.6 pro výpočet parametrů prokládané funkce 6.5 vychází ze vzorců 6.4 (používaných při výpočtu lineárního modelu), do kterých zavádí obecnou funkci jedné proměnné $f(x)$. Prokládaná funkce 6.5 se

skládá z koeficientu $\hat{\beta}_0$ (reprezentující posuv na ose y), koeficientu $\hat{\beta}_1$ (vyjadřující sklon) a zmíněné funkce $f(x)$. Pro dosažení dostačujících výsledků (experimentálně ověřeno srovnáním s přesnými metodami výpočtů) jsou využívány pouze dva koeficienty regresních funkcí (např. u kvadratického modelu $y = a + b \cdot x + c \cdot x^2$ jsou uvažovány pouze koeficienty a a c).

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum x_i}{n} \qquad \hat{\beta}_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2} \tag{6.4}$$

$$Y = \hat{\beta}_0 + \hat{\beta}_1 \cdot f(x) \tag{6.5}$$

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum f(x_i)}{n} \qquad \hat{\beta}_1 = \frac{n \sum f(x_i) y_i - \sum f(x_i) \sum y_i}{n \sum (f(x_i))^2 - (\sum f(x_i))^2} \tag{6.6}$$

Univerzální vzorec v současnosti neprodukuje dostatečně přesné výsledky pro všechny modely a jeho další zpřesnění je cílem navazující práce. Výpočet některých modelů (např. mocninného a exponenciálního) je proto prováděn s využitím konkrétních výpočetních vzorců, které je možné nalézt v [10].

Z vypočítaných regresních modelů je pak nutné vybrat jeden konkrétní, který vyjadřuje závislost mezi proměnnými nejpřesněji. Tuto informaci dokáže vyjádřit tzv. *coefficient of determination* [10], značený jako r^2 a vyjádřený vztahem 6.7. Čím více se hodnota r^2 blíží k 1, tím lépe vyjadřuje zvolený model závislost proměnných.

$$r^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \overline{y})^2} \tag{6.7}$$

6.2 Návrh

Z analýzy problému plyne, že modul musí zvládat výpočet širokého spektra regresních modelů, z nichž některé vyžadují výpočty pomocí specifických vzorců. Z hlediska budoucího vývoje rovněž existuje předpoklad rozšiřování seznamu podporovaných regresních modelů, které mohou mít své naprosto specifické požadavky a úkony (např. rozšíření o nelineární regresi).

Proto musí být architektura modulu navržena tak, aby umožňovala snadné zapojení dalších modelů. Toho lze dosáhnout využitím tabulky modelů, která bude uchovávat jejich specifikace včetně potřebných kroků a příslušných výpočetních operací.

6.2.1 Metody výpočtu

Výpočet všech regresních modelů (a jejich následné porovnání) nad větším množstvím dat může způsobit komplikace z hlediska výkonu nástroje. Navržený modul regresní analýzy řeší tento problém zavedením několika výpočetních technik regresních modelů, které se liší přesností výsledků a rychlostí provádění.

- **Technika úplného výpočtu** provádí úplný výpočet všech modelů přes všechna data. Potenciálně časově náročná metoda, která však vzhledem ke vstupním údajům zajistí přesné výsledky pro všechny modely a nalezne ten nejvhodnější z nich.
- **Iterativní technika** dělí výpočet na několik kroků, kde v každém kroku dochází k výpočtu pouze toho modelu, který má aktuálně nejvyšší hodnotou koeficientu r^2. V úvodní fázi zpracuje technika určitou část bodů (volených náhodně) pro všechny modely a výsledky slouží jako počáteční odhad. Každý další krok zpracuje větší množství náhodně zvolených bodů. Cenou za snížené časové nároky je možnost nalezení lokálního extrému.

- **Technika počátečního odhadu** provede odhad nejvhodnějšího modelu z náhodného vzorku vstupních dat a vybraný model následně kompletně dopočítá. Potenciálně nižší časové nároky než metoda iterativní, avšak za cenu nižší přesnosti a možného nalezení lokálních extrémů.
- **Intervalová metoda** rozdělí vstupní data do specifikovaných intervalů, které lze samostatně analyzovat. Je tak možné odhalit případné odlišnosti mezi regresními modely konkrétních úseků dat (např. změna modelu při větším počtu dat ve struktuře) a tím přesněji analyzovat algoritmus. Díky těmto znalostem je pak možné např. provádět změnu algoritmu v závislosti na množství vstupních dat (řadicí metody v `C#`).
- **Technika bisekce** provádí úplný výpočet a následně se snaží půlením intervalů docílit jemnějšího proložení vzorku dat. Cílem bisekce je vyjádřit určité intervaly přesnějšími modely. Technika je podobná intervalové metodě, ovšem s jistou mírou automatizace hledání vhodných intervalů.

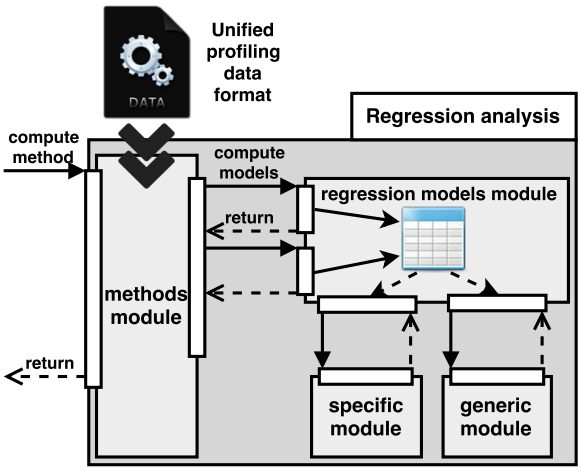
Grafickému znázornění výsledků jednotlivých metod se věnuje experiment 8.1 a jeho grafické výstupy v sekci A.1. V současné implementaci nedochází k automatickému vyhodnocování a volbě nejvhodnější techniky. Stejně tak interpretace výsledků jednotlivých metod je ponechána v kompetenci uživatele.

6.3 Implementace

Implementace modulu regresní analýzy sestává z několika menších částí — konkrétně modulů `methods`, `regression_models`, `specific` a `generic`. Implementace rovněž obsahuje interní pomocné moduly `tools` a `regression_exceptions`, které ale nejsou konceptuálně důležité, a proto nebudou v této sekci popisovány. Obrázek 6.1 schematicky ilustruje komunikaci a závislosti jednotlivých modulů regresní analýzy.

Modul `methods`. Modul slouží jako veřejné rozhraní implementované regresní analýzy. Obsahuje funkce pro výpočet konkrétních metod a technik popsaných v sekci 6.2.1. Každá výpočetní technika deleguje výpočet konkrétních regresních modelů (nebo jejich částí) na modul `regression_models`.

Modul `regression_models`. Spravuje specifikace regresních modelů a implementuje pomocné funkce využívané pro jejich manipulaci. Středobodem modulu je implementace tabulky dle návrhu (viz sekce 6.2), která obsahuje detaily potřebné k výpočtu podporovaných modelů. O samotné výpočty se starají moduly `specific` a `generic`.



Obrázek 6.1: Schéma modulu regresní analýzy, které poskytuje veřejné rozhraní `methods` a které interně deleguje výpočty na moduly `regression_models`, `generic` a `specific`.

Moduly `specific` a `generic`. Moduly jsou výpočetním jádrem jednotlivých regresních modelů. Konkrétně modul `generic` obsahuje generické výpočetní funkce vztahující se k univerzálnímu řešení výpočtů (sekce 6.1) nebo obecně funkce využitelné v širší škále modelů. Naproti tomu modul `specific` implementuje funkce vztahující se ke specifickým regresním modelům (zdůvodnění rovněž v sekci 6.1). Mezi výpočetní funkce patří například výpočet mezivýsledků, navazující výpočet koeficientů modelu a následně výpočet chyby (koeficient r^2). V případě nových regresních modelů stačí využít stávající generické (nebo implementovat nové specifické) výpočty a poté model specifikovat v tabulce modelů.

Obecnost návrhu. Všechny implementované výpočetní funkce musí dodržovat shodné rozhraní v podobě jednotného (avšak flexibilního) formátu argumentů. Díky tomu je možné vytvářet vysoce specifikované výpočetní postupy, které ale mohou být volány jednotným způsobem. Tento přístup zajišťuje rozšiřitelnost nástroje o nové regresní modely.

Kapitola 7

Vizualizace

Tato kapitola se věnuje analýze, návrhu a implementaci modulu pro grafickou vizualizaci výstupů knihovny — tzv. *vizualizéru*. Cílem vizualizéru je umožnit přehlednou a interaktivní vizualizaci profilovacích dat získaných *kolektorem* (viz kapitola 5), jednotlivých výpočetních technik a výsledků regresní analýzy.

7.1 Analýza

Vizualizace výsledků je důležitá především z hlediska uživatele a umožňuje tak efektivní interpretaci nasbíraných a analyzovaných dat — vizuální stránka proto musí být při návrhu modulu v popředí. Z pohledu budoucího vývoje je rovněž nutné brát ohled na snadnou rozšiřitelnost a modifikovatelnost modulu. Následující kritéria ovlivňovala návrh vizualizačního modulu:

1. **Interaktivita, dynamika vizualizace a intuitivnost ovládání.** Možnost interakce s vykreslenými daty a jejich dynamické vlastnosti (např. nápovědy při pohybu myší) jsou z hlediska interpretace výsledků a dojmu uživatele důležité. Pro bezproblémové využití plného potenciálu vizualizačního nástroje je nutné intuitivní a snadné ovládání interaktivních prvků.

2. **Nezávislost na konkrétním vizualizačním nástroji nebo knihovně.** Kvalitní a bohaté grafické zpracování není vždy nutné nebo žádoucí. Modul by proto měl umožnit snadnou volbu mezi více úrovněmi nebo způsoby vizualizace.

7.2 Návrh

Samotné vykreslení grafů je řešeno použitím existující knihovny a komunikační rozhraní navržené v rámci modulu umožňuje jednotný způsob komunikace nezávisle na zvolené knihovně.

7.2.1 Použitá vizualizační knihovna

Pro vykreslení grafů byla zvolena `python` knihovna *Bokeh* [5]. Vykreslení je prováděno s využitím webových technologií, což zajišťuje žádanou interaktivitu a dynamičnost vizualizéru (dále umocněnou volitelným využitím tzv. *Bokeh serveru*). Knihovna nabízí snadnou integraci grafických výstupů přímo na webové stránky a díky tomu lze nástroj v budoucnu

rozšířit o webové rozhraní. Výsledné grafy jsou ukládány jako `HTML` soubory a umožňují export plátna do rastrového grafického formátu.

7.2.2 Komunikační rozhraní

Navržené komunikační rozhraní zastřešuje konkrétní operace s grafickými knihovnami a umožňuje práci jednotným způsobem. Musí implementovat sadu elementárních příkazů, které umožňují základní operace (vytvoření plátna, zobrazení plátna) a vykreslení potřebných grafických primitiv (body, přímky) nezávisle na konkrétní knihovně.

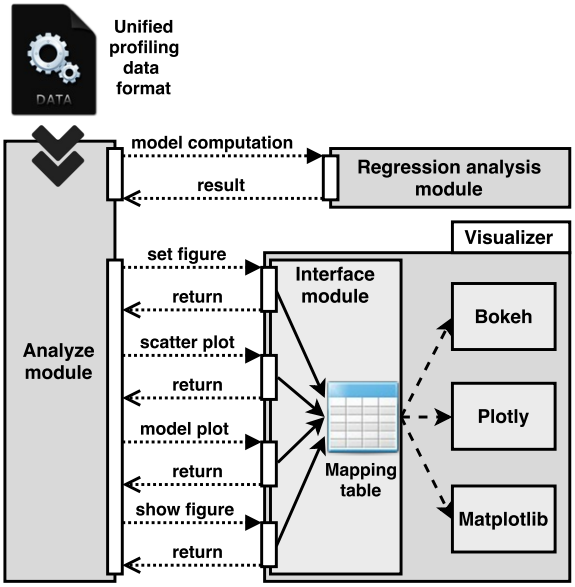
7.3 Implementace

Z implementačního hlediska se vizualizér skládá ze dvou hlavních modulů — `interface` a `bokeh_visualizer`. V této sekci je rovněž popsán modul `analyze` sloužící jako veřejné vysokoúrovňové rozhraní (tzv. *high-level API*) pro ovládání regresní analýzy a vizualizéru.

Modul `interface`. Implementace komunikačního rozhraní ze sekce 7.2.2. Obsahuje čtyři základní příkazy nutné k vykreslování regresních dat:

- `set_figure` slouží k nastavení nového grafického plátna.
- `show_figure` zobrazí a uloží současné grafické plátno.
- `plot_scatter` vykreslí bodový graf.
- `plot_model` vykreslí regresní model.

Modul uchovává tabulku funkcí příslušejících ke každému grafickému modulu a na jejím základě dochází k volbě konkrétní prováděné funkce pro zvolený modul.



Obrázek 7.1: Schéma vizualizéru sestávajícího z rozhraní `interface` a konkrétních grafických modulů (v současnosti implementován jen *Bokeh*). Modul `analyze` slouží k propojení modulů regresní analýzy a vizualizéru.

Modul `bokeh_visualizer`. Implementuje funkce vyžadované vizualizačním rozhraním `interface` pomocí grafické knihovny *Bokeh*. V současnosti se jedná o jediný implementovaný vizualizér, avšak díky komunikačnímu rozhraní lze snadno integrovat další podobné knihovny (za cenu méně bohatšího grafického rozhraní a umožňující například export do vektorových formátů atd.).

Modul `analyze`. Zastřešuje a řídí komunikaci modulu regresní analýzy s vizualizačním rozhraním *interface*. Poskytuje vysokoúrovňové rozhraní pro jednotlivé regresní výpočetní metody (sekce 6.2.1). Jedná se o volitelný modul, jež není potřeba využívat, avšak který centralizuje analyzační a vizualizační modul pro snadnější používání.

Kapitola 8

Experimentální ověření vytvořeného nástroje

Tato kapitola demonstruje praktické využití implementovaného nástroje a ověření jeho funkčnosti sérií experimentů nad reálnými daty. Pro každý z experimentů bylo vytvořeno množství C/C++ programů, které implementují profilované algoritmy nebo operace datových struktur. Z těchto programů následně kolektor vytvořil profil (ve tvaru jednotného profilovacího formátu dat, viz sekce 4.2), jež byl v některých případech dále podroben regresní analýze.

Výstupem experimentů jsou komentované grafy produkované implementovaným vizualizérem. Kapitola se věnuje popisu jednotlivých experimentů, jejich cílů a vyhodnocení dosažených výsledků. Grafické výstupy experimentů jsou z důvodu svého rozsahu umístěny do přílohy A.

8.1 Demonstrace výpočetních metod regresní analýzy

Cílem experimentu je ověřit, zda nástroj dokáže korektně odhadnout výpočetní složitosti algoritmů a operací datových struktur s využitím implementovaných výpočetních metod (sekce 6.2.1) regresní analýzy. Pro ověření jsou výstupní odhady nástroje srovnány s teoretickými složitostmi vstupních algoritmů.

Operace	Teoretický odhad	Výstup nástroje
`Red-Black Tree insert`	$\Theta(\log n)$	$\Theta(\log n)$
`Red-Black Tree search`	$\Theta(\log n)$	$\Theta(\log n)$
`Insertion-sort`	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
`Stack search`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`Skiplist search`	$\Theta(\log n)$	$\Theta(n)$
`Degradovaný Skiplist search`	$\Theta(n)$	$\Theta(n)$

Tabulka 8.1: Shrnutí výsledků experimentu.

Metoda úplného výpočtu analyzuje operace vložení a vyhledání prvku nad datovou strukturou *červeno-černého stromu*[1] (stručně popsán jako jedna z variant binárního vyhledávacího stromu v sekci 3.3). Pomocí iterativní techniky byl analyzován vstup řadicího

[1] C++ implementace převzata z https://gist.github.com/windoro/1c8e458e9ab5e8e4c5bb

algoritmu *Insertion-sort*, jehož teoretický rozbor byl proveden v sekci 2.2.5 a díky tomu lze srovnat oba přístupy odhadu časových složitostí. Došlo k rozšíření datové struktury zásobníku o operaci vyhledání prvku (lineární průchod směrem od vrcholu zásobníku), na které byla demonstrována technika počátečního odhadu. Intervalová metoda je předvedena na jednom z výstupů experimentu 8.2.

Výsledky téměř všech zkoumaných algoritmů a operací se shodují s teoretickými předpoklady a experiment tak lze označit za úspěšný. Jediný rozdíl nastal v případě využití intervalové techniky pro výškově omezený skip list, kde se nepodařilo zvolit ideálně dlouhý interval pro demonstraci logaritmické složitosti. Podrobnější rozbor lze nalézt u obrázku A.5. Tabulka 8.1 obsahuje souhrn výsledků pro porovnání. Grafické výstupy s komentářem se nacházejí v sekci A.1.

8.2 Vliv parametrů na výkon skip list *search* operace

Experiment podrobněji analyzuje dopady zvolených parametrů na výkon operace vyhledání posledního prvku ve skip listu. Mezi zkoumané parametry patří maximální počet výškových úrovní a pravděpodobnost vytvoření nové výškové úrovně pro daný prvek. Cílem je demonstrovat možné využití nástroje pro praktické porovnání pravděpodobnostní struktury, jejíž chování ovlivňují uživatelem zvolené parametry.

Postupně je analyzován skip list[2] s neomezeným počtem výškových úrovní (tzn. není dosaženo maximální povolené výšky 50 pro žádný vložený prvek), vysokým počtem úrovní (13) a nízkým počtem úrovní (3). Všechny předchozí verze skip listu pracují s pravděpodobností 50% pro vytvoření nové výškové úrovně. Závěrem je provedeno porovnání skip listů s nízkým počtem výškových úrovní (3), avšak s rozdílnou pravděpodobností vytvoření nové výškové úrovně (50% a 10%).

	$L:\infty,\ P:50\%$	$L:13,\ P:50\%$	$L:3,\ P:50\%$	$L:3,\ P:10\%$
`Složitost`	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Tabulka 8.2: Shrnutí výsledků pro verze skip listu s různým počtem výškových úrovní L a různou pravděpodobností vytvoření nové výškové úrovně P.

Výsledky experimentu odpovídají očekávání. Se snižujícím se počtem úrovní dochází k postupné linearizaci procesu vyhledání posledního prvku, čímž může dojít (v závislosti na parametru pravděpodobnosti vytvoření nové výškové úrovně) ke ztrátě výhody skip listu v podobě rychlého vyhledávání. Některé verze skip listu vykazují lineární složitost vyhledání posledního prvku, avšak důležitou roli na celkovém času vyhledání hraje právě koeficient $\hat{\beta}_1$. V závislosti na jeho hodnotě se např. liší doba operace v posledních dvou případech až stonásobně. Shrnutí výsledků je v tabulce 8.2 a komentované grafické výstupy lze nalézt v sekci A.2.

8.3 Srovnání seznamu a skip listu z hlediska spotřeby zdrojů

Experiment se zabývá srovnáním datových struktur seznamu a skip listu. Struktury jsou porovnávány z hlediska paměťových nároků a jejich vlivu na rychlost operace vyhledání

[2] Implementace převzata z http://www.cs.yale.edu/homes/aspnes/classes/223/examples/trees/skiplist/

posledního prvku. Informace o spotřebě paměti jsou získány za pomoci nástroje, který je výsledkem práce [18].

Nejprve jsou srovnány paměťové a časové nároky seznamu se skip listem, který není omezen z hlediska počtu výškových úrovní a pravděpodobnost jejich zvýšení je rovna 50%. Následně je provedeno srovnání s paměťově optimalizovanou verzí skip listu, jež obsahuje maximálně pět výškových úrovní a k navýšení úrovně dochází s pravděpodobností 10%.

Výsledky experimentu dopadly dle očekávání. Bylo demonstrováno, že za cenu vyšších paměťových nároků lze v případě skip listu urychlit proces vyhledání prvku. Množství spotřebované paměti však nemusí být přímo úměrné získanému zrychlení — správná volba parametrů dokáže zajistit nízký nárůst paměťových nároků za cenu výrazného zrychlení operace (jedná se sice o lineární model, avšak jeho koeficient b je téměř konstantní, viz grafické výstupy). Konkrétní výsledky lze nalézt v tabulce 8.3 shrnující výsledky experimentu, jehož grafické výstupy lze nalézt v sekci A.3.

Operace	Časové nároky	Spotřebovaná paměť
`List search`	$\mathcal{O}(n)$	16000 B / 1000 prvků
`Skip list search`	$\Theta(\log n)$	24048 B / 1000 prvků
`Optimalizovaný skip list search`	$\Theta(n)$	17024B / 1000 prvků

Tabulka 8.3: Shrnutí výsledků porovnání časových a paměťových nároků seznamu a skip listu. Ve všech případech je vyhledáván poslední prvek ve struktuře. Jiná notace v případě skip listu je způsobena jiným scénářem, který způsobuje nejhorší případ.

8.4 Výkon algoritmu Quicksort v závislosti na volbě pivotu

Cílem experimentu je demonstrovat závislost časové složitosti řadicího algoritmu *Quicksort*[3] na způsobu, jakým je vybrán pivot a na dopadu této volby pro různé vstupy.

V první sérii experimentů je vybrán jako pivot okrajový prvek řazené posloupnosti a ve druhé sérii se jedná o prostřední prvek posloupnosti. Série sestává ze spuštění algoritmu nad seřazeným, opačně seřazeným a náhodně uspořádaným vstupem.

Vstup	Pivot - okraj	Pivot - střed
`Seřazený`	$\mathcal{O}(n^2)$	$\Theta(n)$
`Opačně seřazený`	$\mathcal{O}(n^2)$	$\Theta(n)$
`Náhodně uspořádaný`	$\Theta(n)$	$\Theta(n)$

Tabulka 8.4: Odhady časových složitostí jednotlivých variant a vstupů algoritmu. Nástroj v současnosti neposkytuje implementaci linearitmického modelu $(n \cdot \log n)$ a z důvodu velké podobnosti tak dochází k odhadu lineárního modelu.

Výsledné odhady (tabulka 8.4) produkované nástrojem přibližně odpovídají teoretickým předpokladům, kdy pro seřazené posloupnosti a okrajový pivot vzniká nejhorší případ vstupu. Grafické výstupy experimentu se nacházejí v sekci A.4.

[3] Implementace převzata z http://www.geeksforgeeks.org/iterative-quick-sort/

8.5 Způsoby srovnávání výkonu algoritmů

Poslední experiment demonstruje dostupné způsoby porovnání časových složitostí algoritmů. Nástroj implementuje dvě metody — *absolutní* a *relativní* srovnání. První metoda vykresluje profilovací data všech algoritmů ve formě bodového grafu do stejného plátna a umožňuje tak porovnat absolutní rozdíly mezi body jednotlivých algoritmů. Relativní srovnání umožňuje porovnat dva algoritmy pomocí jednoho bodového grafu, kde souřadnice x a y značí časové údaje prvního, respektive druhého algoritmu pro stejnou velikost struktury. Časově efektivnější algoritmus pro danou velikost je určen odchylkou výsledného bodu od přímky $y = x$.

Srovnání je provedeno nad řadicími algoritmy *Quicksort*, *Heapsort*[4] a *Insertion-sort* s náhodně uspořádaným vstupním polem. Interpretace porovnání algoritmů je ponechána na uživateli, a proto není v tomto případě využita regresní analýza.

Výsledky porovnání zmíněných tří algoritmů odpovídají teoretickým předpokladům — Insertion-sort je ze všech tří algoritmů nejpomalejší a Quicksort s Heapsortem dosahují přibližně stejných výsledků. Grafické výstupy obou srovnávacích metod jsou v sekci A.5.

[4] Implementace převzata z https://www.algoritmy.net/article/17/Heapsort

Kapitola 9

Závěr

Cílem této práce bylo navrhnout a implementovat prakticky použitelný nástroj, který nalezne využití při profilaci datových struktur v jazyce C/C++. Výsledný nástroj umožňuje efektivní měření potřebných profilačních dat, jejich analýzu a především kvalitní grafický výstup, který usnadní interpretaci výsledků — odhadů časových složitostí jednotlivých operací a algoritmů nad datovými strukturami.

Úvod práce se zaměřuje na rozbor odborných a teoretických podkladů řešeného problému. Konkrétně je představena oblast složitosti algoritmů (asymptotická, amortizovaná) a současný přístup k jejich odvození, který sestává z teoretické analýzy zdrojového kódu. Dále jsou představeny vybrané datové struktury, které jsou běžně používané v praxi.

Byl navržen profilační nástroj tvořený třemi vzájemně spolupracujícími moduly, které mohou být nezávisle modifikovány a rozšiřovány. Každý z těchto modulů obstarává řešení některého z podproblémů — sběr dat, jejich analýzu a vizualizaci výsledků. Nástroj byl vyvíjen jako součást projektu Perun, který si klade za cíl integrovat spektrum profilovacích nástrojů a automatizovat jejich využití při vývoji projektů.

Na základě návrhu byl naimplementován prototyp nástroje, který byl následně podroben experimentálnímu ověření. Z výsledků experimentů je patrné, že nástroj produkuje převážně korektní výstupy, jež se shodují s teoretickými předpoklady časových složitostí. Bylo demonstrováno množství případů, ve kterých může nástroj nalézt praktické uplatnění a tím přispět k rozšíření současných možností v oblasti profilace a výkonnostní analýzy.

Navazující práce se bude zaměřovat na dokončení integrace do nástroje Perun a zpřesnění produkovaných výsledků díky využití důmyslnějších statistických metod a postupů. Jedním z dalších směrů je rovněž rozšíření množiny podporovaných regresních modelů, výpočetních metod, vizualizačních knihoven a technik pro sběr profilačních dat. Cílem dalšího vývoje je vytvoření sofistikovaného nástroje, který by našel skutečně praktické využití při časové profilaci napříč množstvím vyvíjených programů.

Tato práce byla přijata do sborníku studentské konference Excel@FIT 2017 (příspěvek č. 55), kde byly prezentovány dosažené výsledky společně s kolegou R. Podolou [18].

Literatura

[1] Aho, A. V.; Hopcroft, J. E.; Ullman, J.: *Data Structures and Algorithms*. Addison-Wesley, 1983, ISBN 0201000237.

[2] *Algoritmy.net, Spojový seznam*. [Online; navštíveno 10.2.2017].
URL https://www.algoritmy.net/article/24/Spojovy-seznam

[3] Atkey, R.: *Amortised Resource Analysis with Separation Logic. Logical Methods in Computer Science*, ročník 7, 2011: s. 1–33, ISSN 1860-5974.

[4] *Big-O Algorithm Complexity Cheat Sheet*. [Online; navštíveno 10.5.2017].
URL http://bigocheatsheet.com/

[5] *Bokeh: Python interactive visualization library*. [Online; navštíveno 1.5.2017].
URL http://bokeh.pydata.org/

[6] *Chrono, C++11 time library*. [Online; navštíveno 14.4.2017].
URL http://www.cplusplus.com/reference/chrono/

[7] *Clang 5 documentation: Command line argument reference*. [Online; navštíveno 14.4.2017].
URL https://clang.llvm.org/docs/ClangCommandLineReference.html

[8] *CMake homepage*. [Online; navštíveno 17.4.2017].
URL https://cmake.org/

[9] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to Algorithms*. MIT Press, 1990, ISBN 0-262-03293-7.

[10] Devore, J. L.: *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 8 vydání, 2011, ISBN 0-8400-6827-1.

[11] Fiebrink, R.: *Amortized Analysis Explained*. 2007, [Online; navštíveno 11.1.2017].
URL https://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf

[12] Fiedor, T.: *Perun: Lightweight Performance Control System*. [Online; navštíveno 3.5.2017].
URL https://github.com/tfiedor/perun

[13] *GCC: Options for Code Generation Conventions*. [Online; navštíveno 10.1.2017].
URL https://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Code-Gen-Options.html

[14] Honzík, J. M.: *Algoritmy, studijní opora IAL*. Verze 14 N, Fakulta Informačních Technologií, Vysoké Učení Technické v Brně, 2014.

[15] *JSON: JavaScript Object Notation*. [Online; navštíveno 10.5.2017].
URL http://www.json.org/index.html

[16] *KCachegrind: Call Graph Viewer*. [Online; navštíveno 12.5.2017].
URL https://kcachegrind.github.io/html/Home.html

[17] *Microsoft .NET Framework 4.6 and 4.5 documentation*. [Online; navštíveno 8.2.2017].
URL https://msdn.microsoft.com/en-us/library/w0x726c2(v=vs.110).aspx

[18] Podola, R.: *Knihovna pro profilování a vizualizaci spotřeby paměti programů C/C++*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2017, vedoucí práce Fiedor Tomáš.

[19] Pugh, W.: *Skip Lists: A Probabilistic Alternative to Balanced Trees*. *Communications of the ACM*, ročník 33, 1990: s. 668–676, doi:10.1145/78973.78977.

[20] *Readelf - GNU Binary Utility*. [Online; navštíveno 18.4.2017].
URL https://sourceware.org/binutils/docs/binutils/readelf.html

[21] Tarjan, R. E.: *Amortized Computational Complexity*. *SIAM Journal on Algebraic Discrete Methods*, ročník 6, č. 2, 1985: s. 306–318, ISSN 0196-5212.

[22] *Valgrind's Tool Suite*. [Online; navštíveno 10.4.2017].
URL http://valgrind.org/info/tools.html

[23] Wirth, N.: *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976, ISBN 0-13-022418-9.

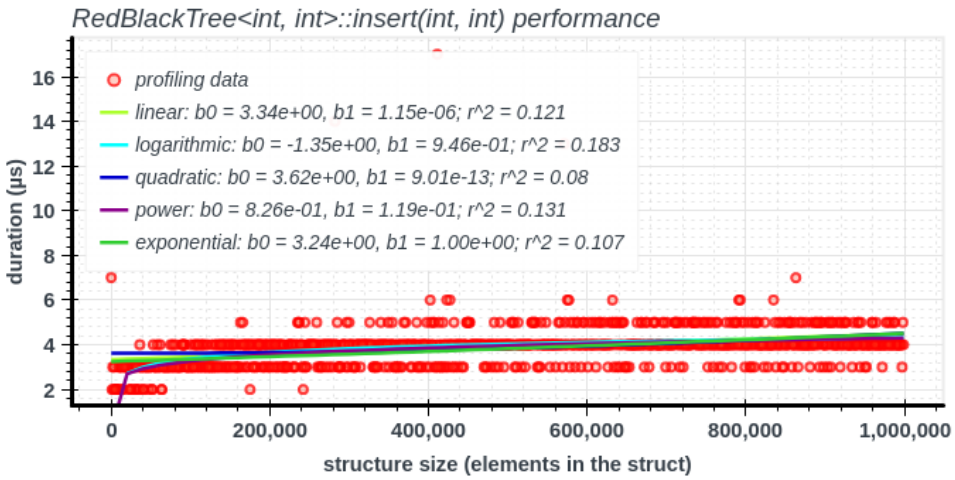
[24] Černý, J.: *Časová složitost v průměrném případě*. 2016, [Online; navštíveno 10.1.2017].
URL http://algoritmy.eu/zga/casova-slozitost/v-prumernem-pripade/

Přílohy

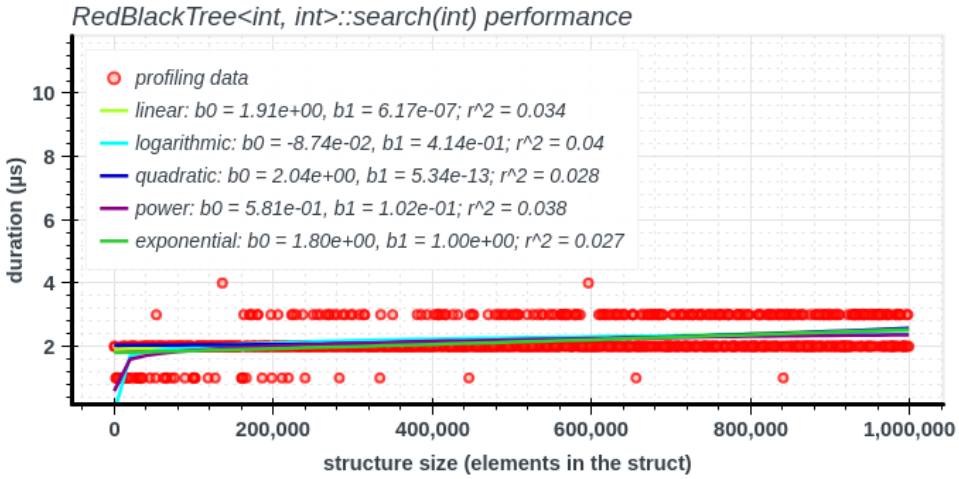
Příloha A

Grafické výstupy experimentálního ověření

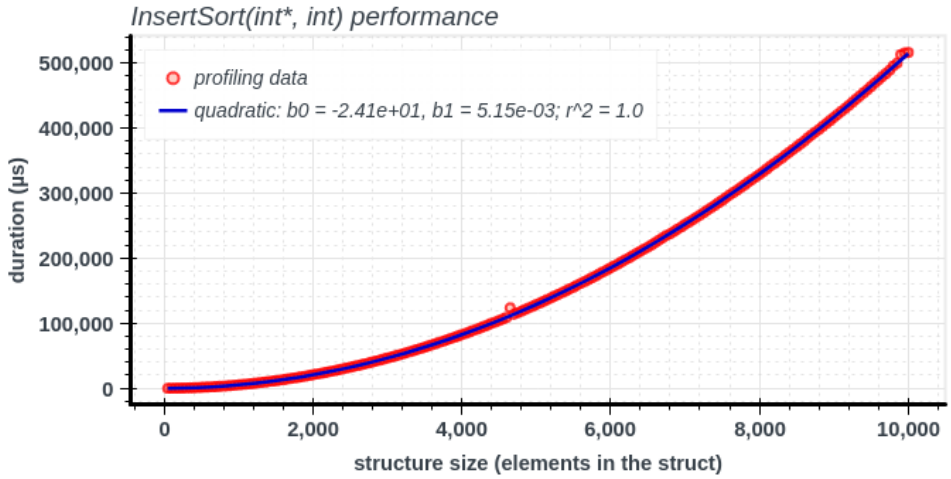
A.1 Demonstrace výpočetních metod regresní analýzy



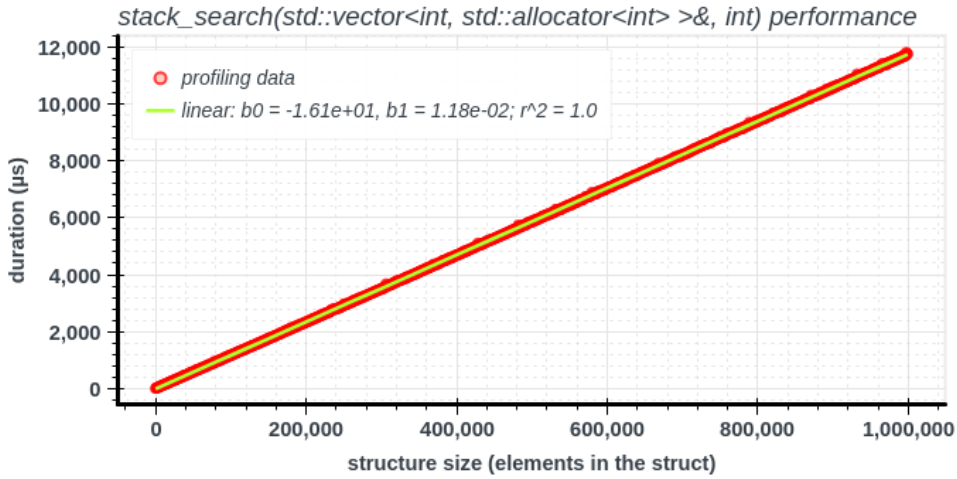
Obrázek A.1: Operace vložení prvku do struktury červeno-černého stromu ($\Theta(\log n)$).



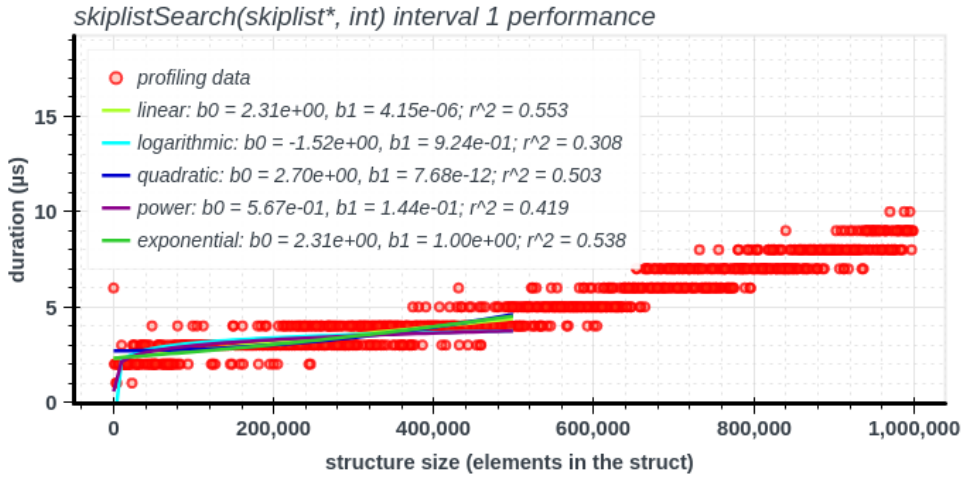
Obrázek A.2: Operace vyhledání prvku ve struktuře červeno-černého stromu ($\Theta(\log n)$).



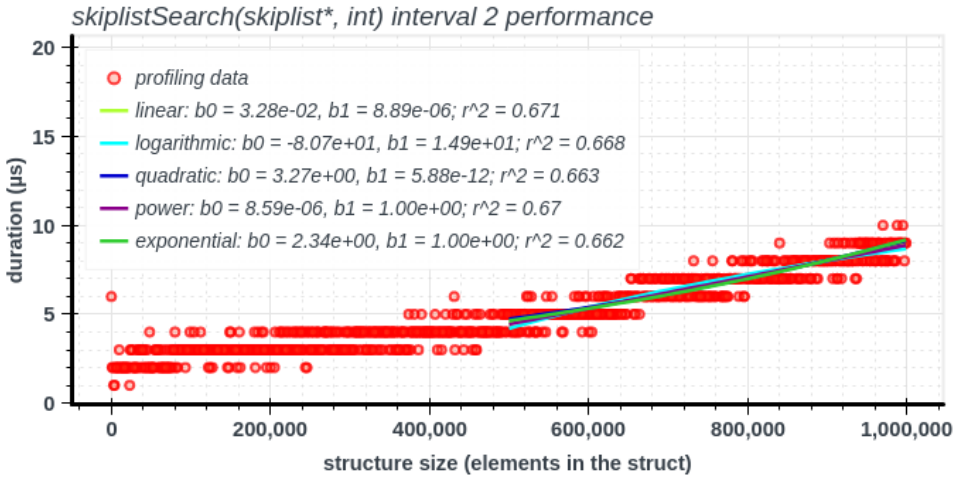
Obrázek A.3: Insertion-sort algoritmus nad vstupem v nejhorším případě ($\mathcal{O}(n^2)$).



Obrázek A.4: Rozšíření zásobníku o operaci vyhledání posledního prvku ($\mathcal{O}(n)$).

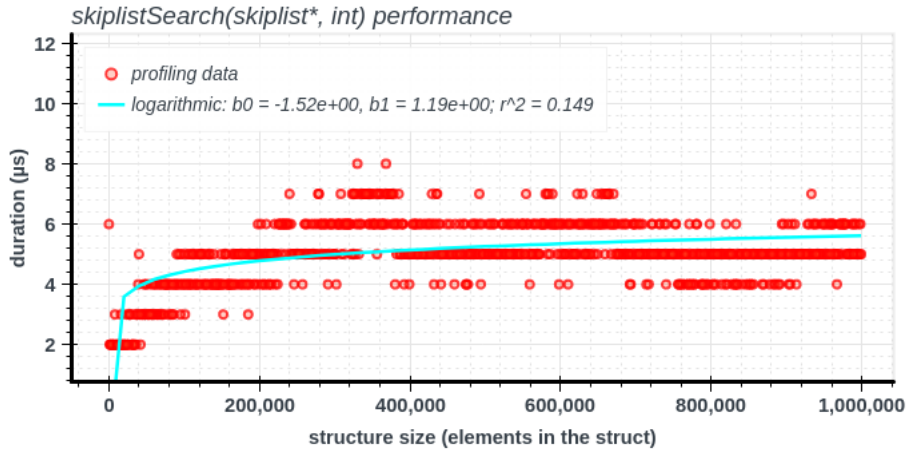


Obrázek A.5: Analýza prvního intervalu operace vyhledání prvku ve struktuře skiplist se 13 výškovými úrovněmi. Interval nedosahuje očekávaného výsledku $\Theta(\log n)$ především kvůli nižšímu počtu dat a příliš dlouhému intervalu. Logaritmický trend je však lidským pozorovatelem postřehnutelný.

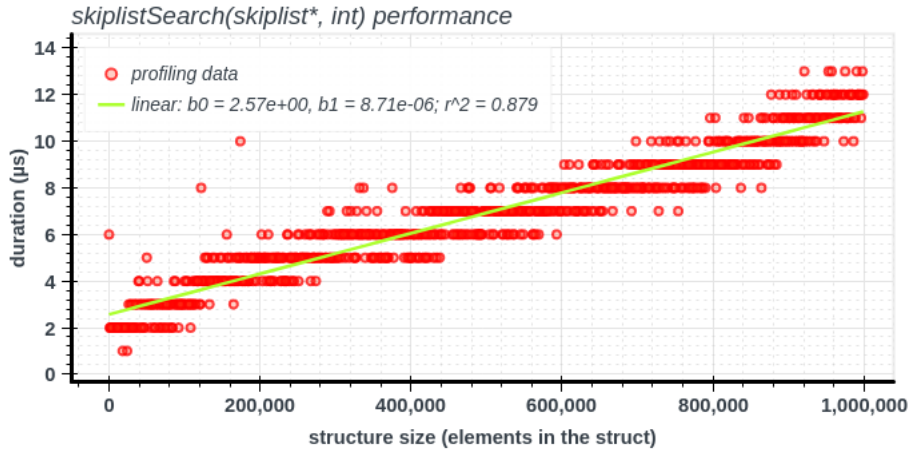


Obrázek A.6: Druhá polovina intervalu již dosahuje očekávané složitosti $\Theta(n)$. Skiplist dosáhl své maximální výškové úrovně a hledání posledního prvku tak obnáší stále větší počet průchodů (i v nejvyšší úrovni).

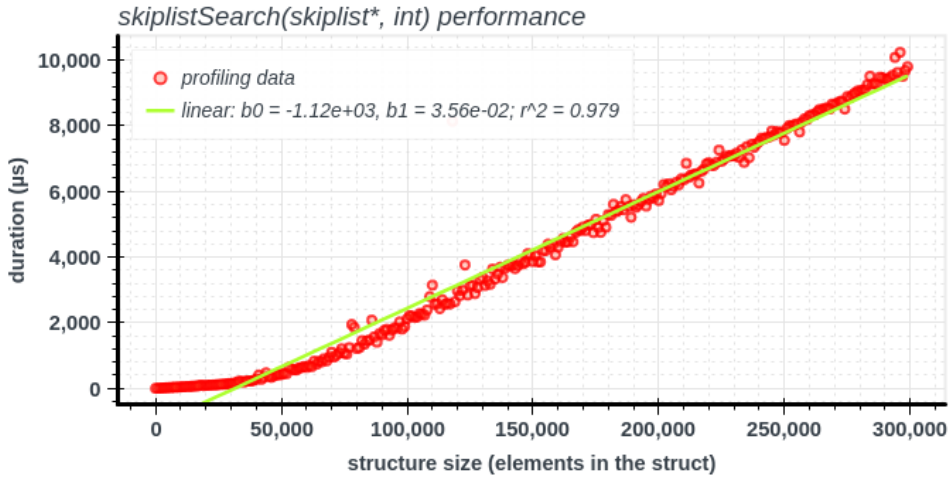
A.2 Vliv parametrů na výkon skip list *search* operace



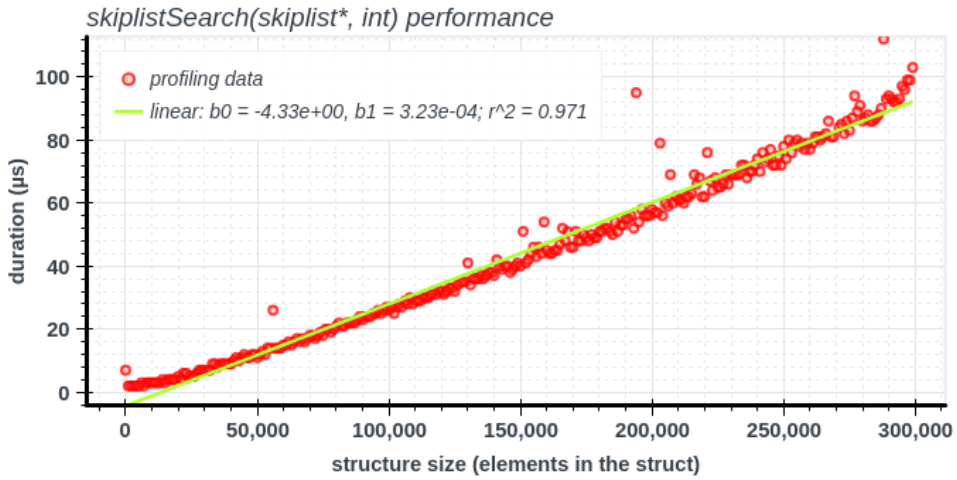
Obrázek A.7: Skip list s 50 úrovněmi vykazuje očekávané chování $\Theta(\log n)$ při vyhledávání.



Obrázek A.8: V případě 13 úrovní je patrný zlom z logaritmického na lineární model.

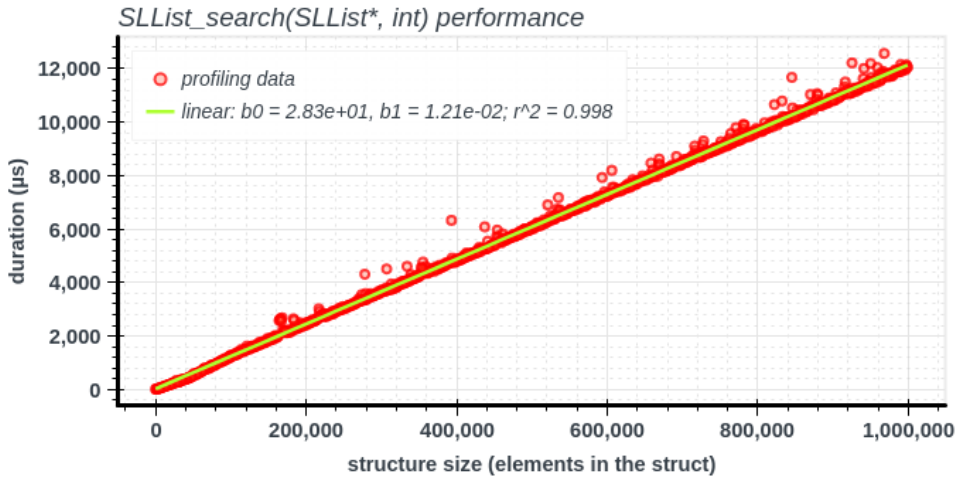


Obrázek A.9: Skip list se třemi úrovněmi a 50% šancí na jejich zvyšování vykazuje už při nízkém počtu prvků ve struktuře zlom na lineární model. Ztrácí se tak podstatná výhoda skip listu v podobě rychlého vyhledávání.

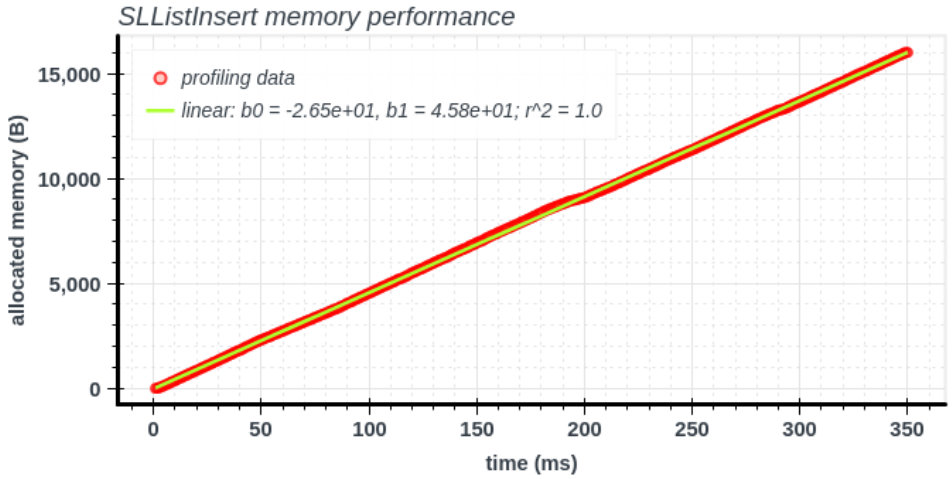


Obrázek A.10: Stejně jako v předchozí demonstraci má skip list maximálně tři výškové úrovně. Došlo však ke snížení pravděpodobnosti vytvoření nové úrovně na 10%. Při takto zvolených parametrech lze pozorovat nárůst efektivity vyhledání posledního prvku až stonásobně oproti předchozí demonstraci.

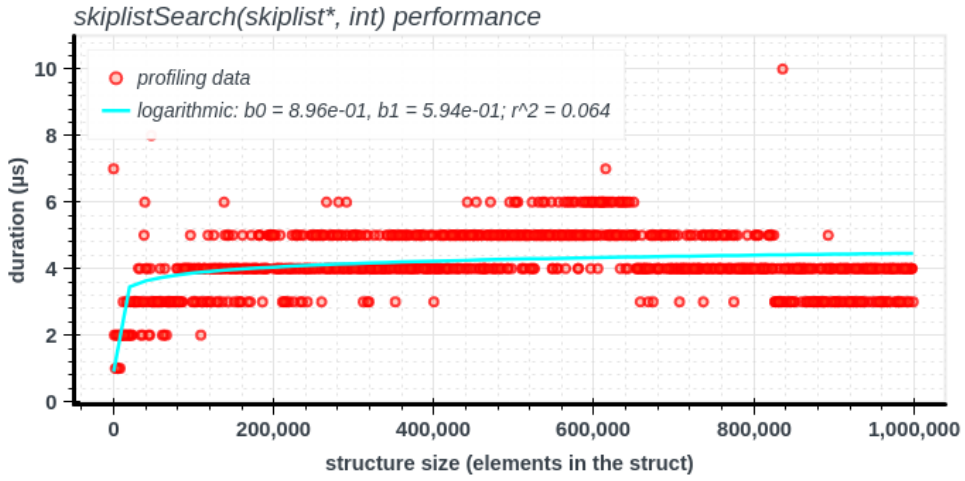
A.3 Srovnání seznamu a skip listu z hlediska spotřeby zdrojů



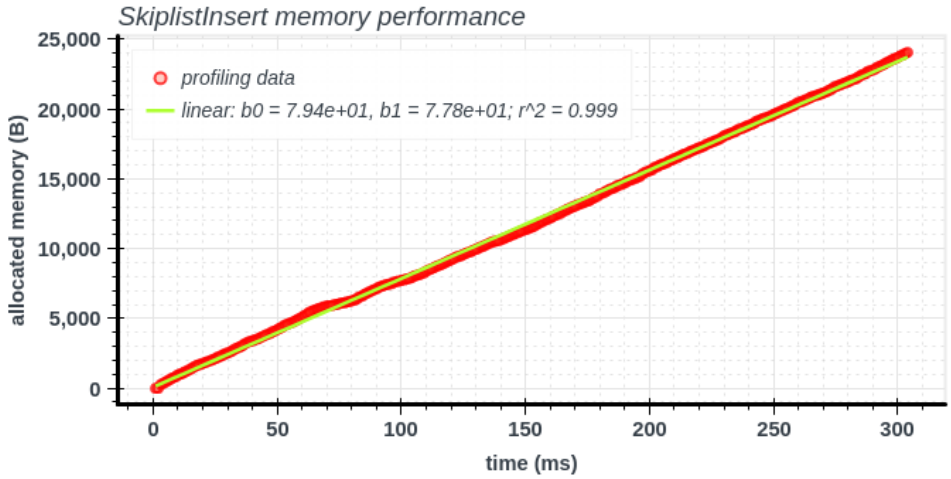
Obrázek A.11: Vyhledání posledního prvku v seznamu dosahuje očekávané složitosti $\mathcal{O}(n)$.



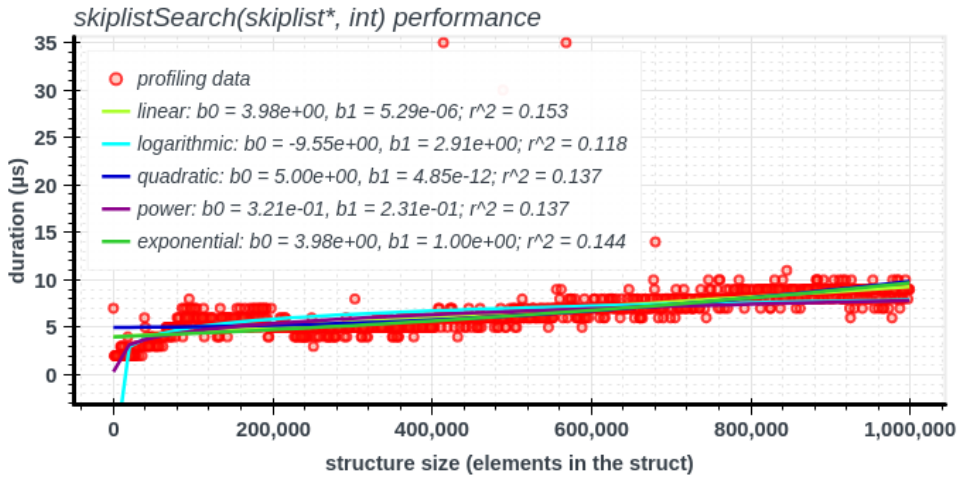
Obrázek A.12: Odpovídající paměťové nároky stejného seznamu pro 1000 vložených prvků.



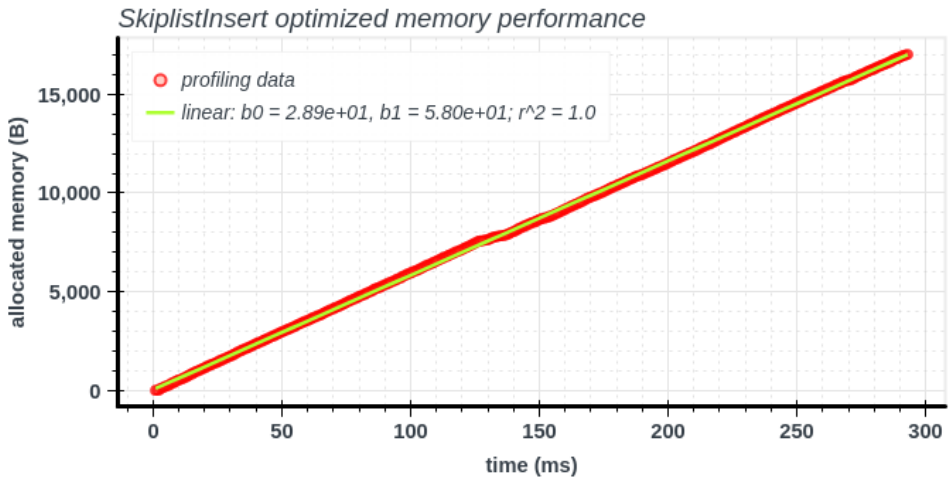
Obrázek A.13: Vyhledání posledního prvku ve skip listu s 50 úrovněmi ($\Theta(\log n)$).



Obrázek A.14: Paměťové nároky pro 1000 vložených prvků u skip listu jsou oproti seznamu o polovinu vyšší (24048 B oproti 16000 B).

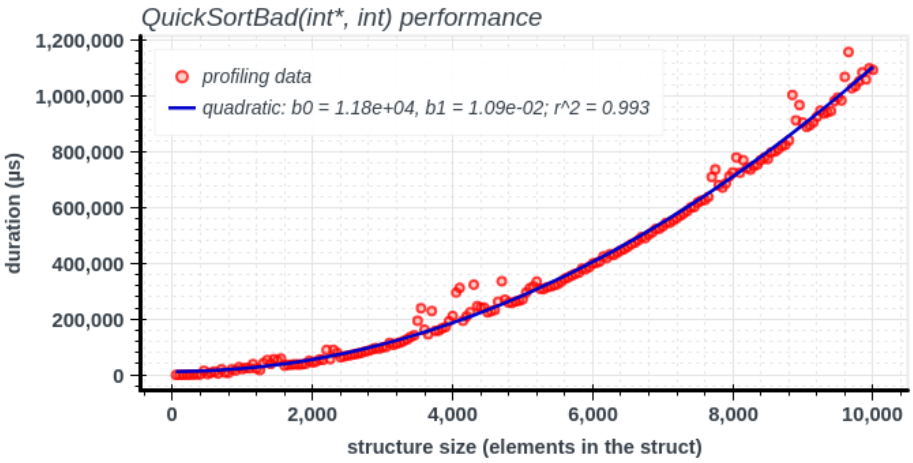


Obrázek A.15: Vyhledání posledního prvku u skip listu s 5 úrovněmi a 10% šanci na zvýšení sice změní složitost na lineární, avšak s velmi nízkým koeficientem b.

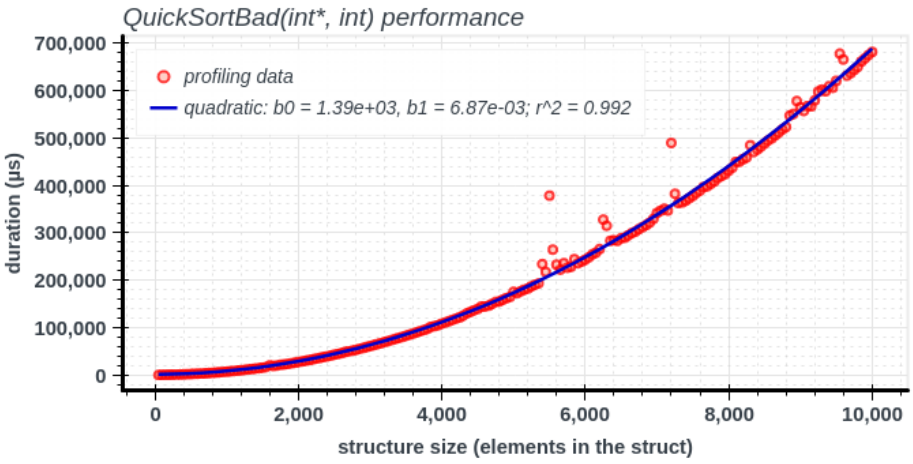


Obrázek A.16: Spotřeba paměti je oproti předchozí verzi skip listu nižší (17024 B oproti 24048 B) při zachování podobných časových vlastností vyhledání posledního prvku.

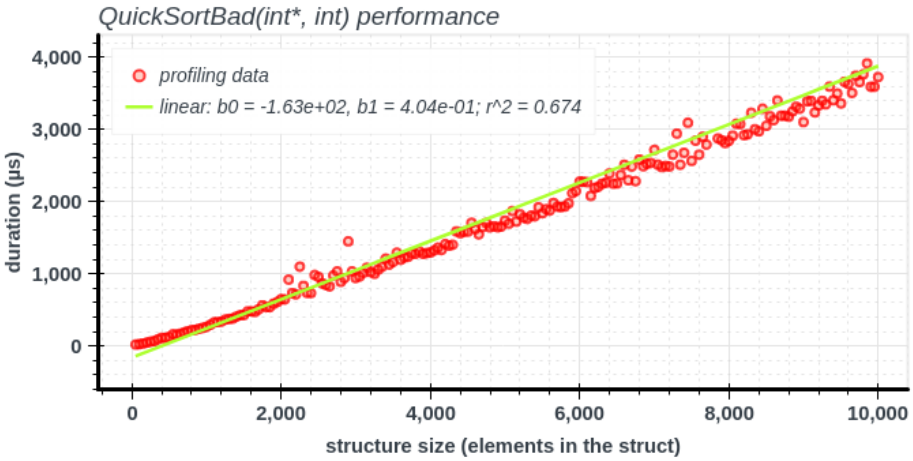
A.4 Výkon algoritmu Quicksort v závislosti na volbě pivotu



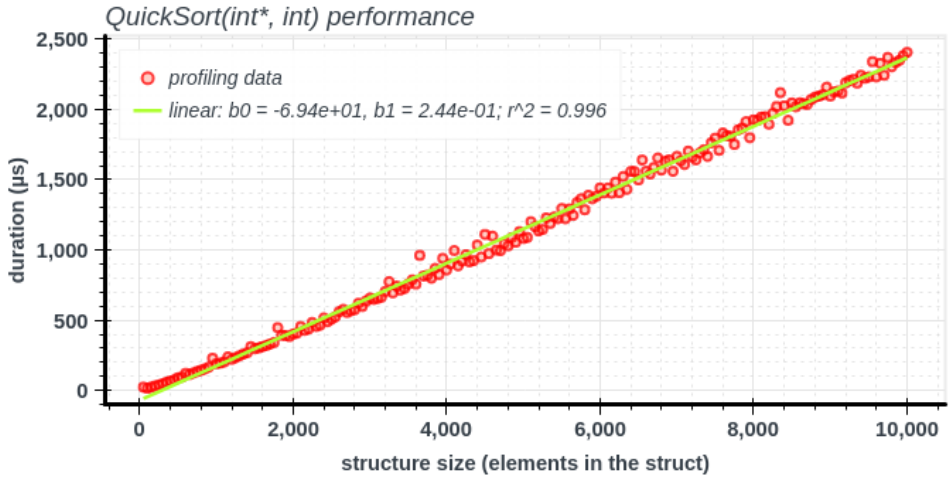
Obrázek A.17: Seřazená posloupnost představuje pro algoritmus Quicksort, u kterého slouží okrajový prvek jako pivot, nejhorší případ vstupu ($\mathcal{O}(n^2)$).



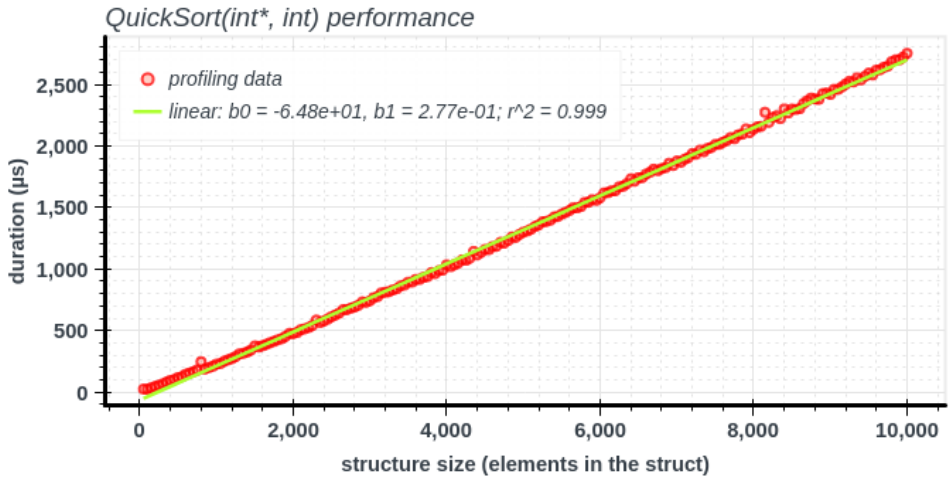
Obrázek A.18: Pro daný pivot je i opačně seřazená posloupnost nejhorším vstupem ($\mathcal{O}(n^2)$).



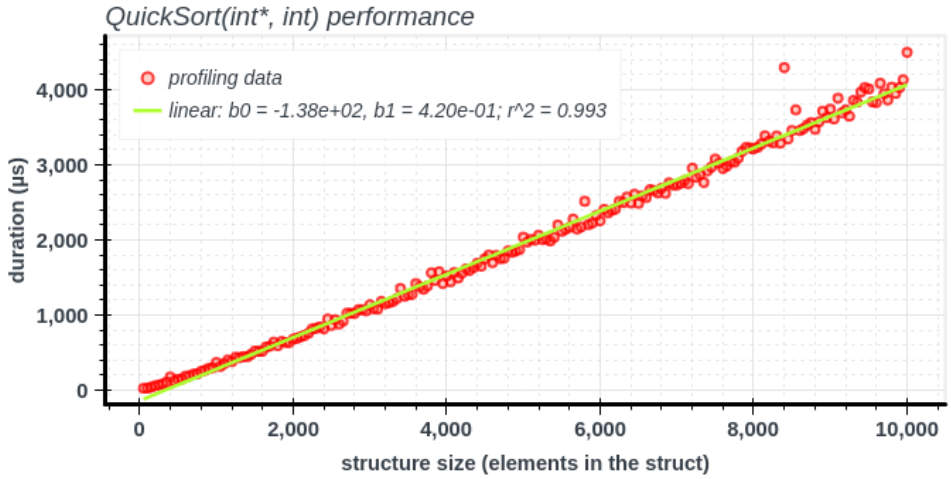
Obrázek A.19: Náhodně uspořádaná sekvence na vstupu již nevytváří pro okrajový pivot nejhorší případ vstupu (není implementován linearitmický regresní model, viz sekce 8.4).



Obrázek A.20: Pokud je pivot zvolen jako prostřední prvek řazené posloupnosti, tak Quicksort nad seřazeným vstupem již nevykazuje nejhorší chování. Ve skutečnosti se jedná o optimální případ, kdy pivot dělí řazenou sekvenci na poloviny.

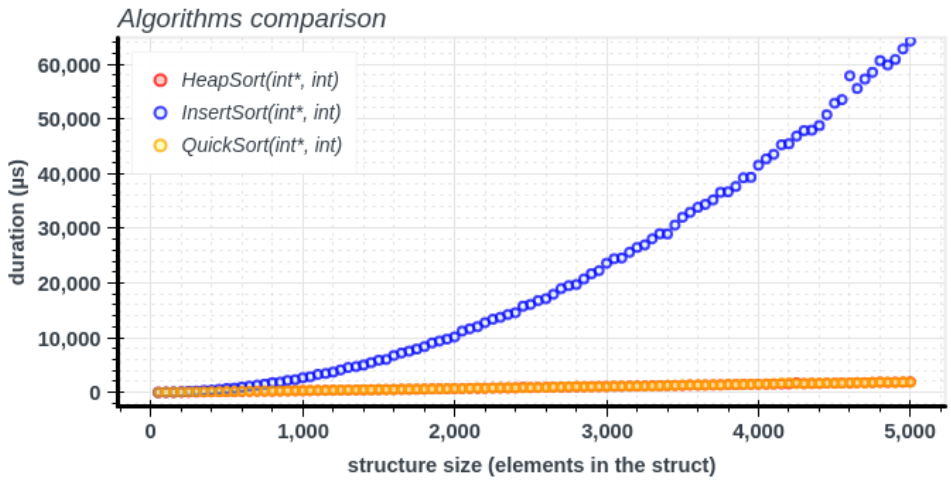


Obrázek A.21: Opačně seřazená vstupní posloupnost je rovněž optimálním vstupem.

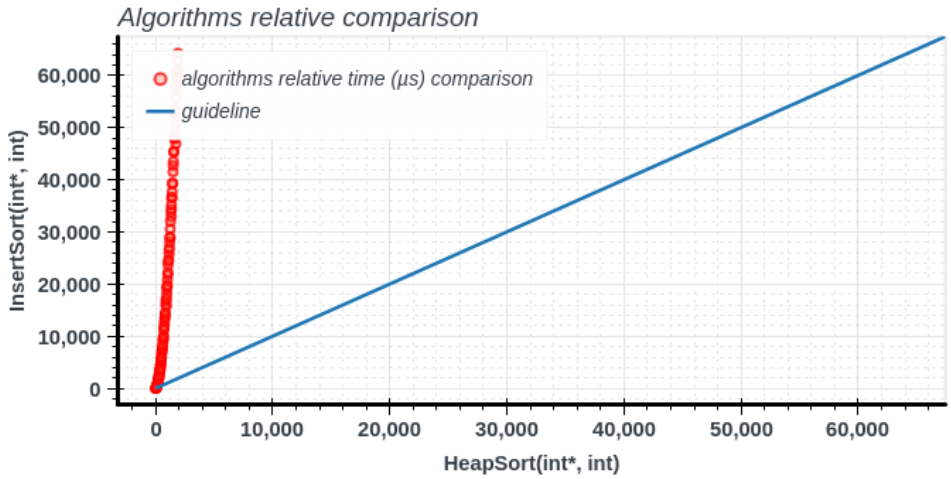


Obrázek A.22: Rozdílná volba pivotu neměla na náhodně uspořádaný vstup vliv, a proto jsou výsledky téměř shodné.

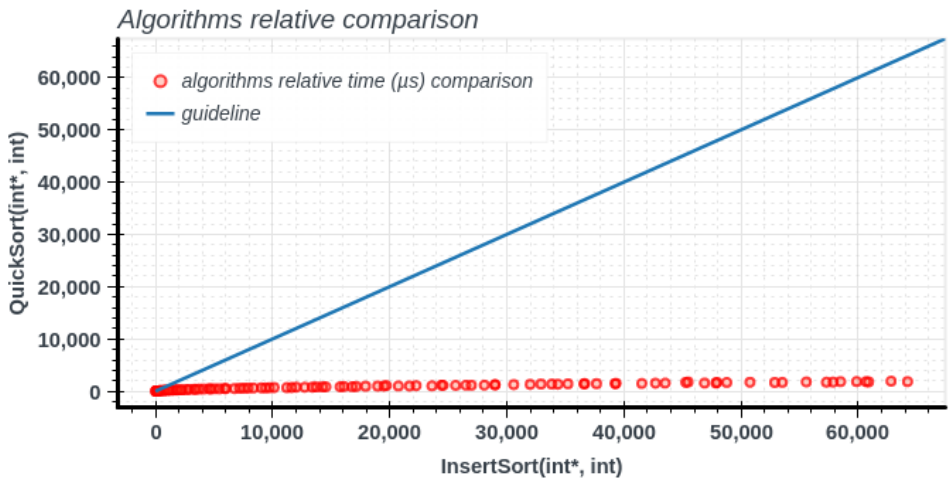
A.5 Způsoby srovnávání výkonu algoritmů



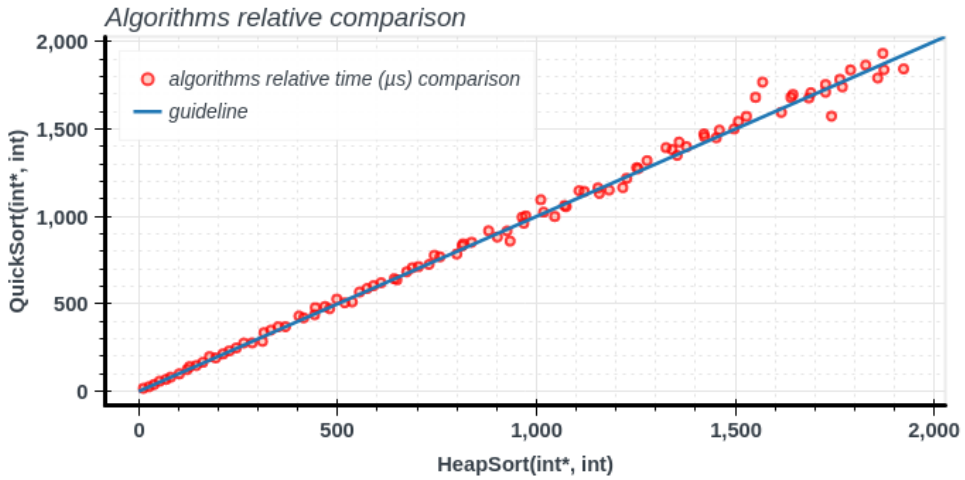
Obrázek A.23: Ukázka absolutního srovnání tří algoritmů — Heapsort, Quicksort a Insertsort — nad shodným vstupem, kterým jsou náhodně uspořádané posloupnosti. Odlišnost Insertsoru je z tohoto srovnání patrná, avšak detailnější porovnání zbylých dvou algoritmů není kvůli velkému měřítku a podobným výsledkům možné.



Obrázek A.24: Relativní srovnání algoritmu Insertsort a Heapsort vzhledem k jejich době běhu. Porovnání je prováděno nad stejnými profilačními údaji jako v předchozím případě. Podrobnější popis časové relativního srovnání algoritmů se nachází v sekci 8.4.



Obrázek A.25: Podobně jako v předchozí ilustraci se jedná o relativní srovnání Quicksort a Insertsort algoritmů.



Obrázek A.26: Relativní srovnání Quicksort a Heapsort algoritmů. S pomocí relativního srovnání lze detailněji studovat chování podobných algoritmů z hlediska jejich rychlosti.