Aufgabe zu HashSets

Erik Imgrund, TINF19B1

26. Mai 2020

1 Grundlegendes

1.1 Verwendete Verfahren

Zur Kollisionsauflösung wurden Direct Chaining, Separate Chaining, Linear Probing, Quadratic Probing, Triangular Probing und ein Coalesced Table implementiert. Als Hashfunktionen wurden einmal der simple ModHash und MulHash aus der Vorlesung implementiert, sowie ein gebräuchlicher XorShiftHash. Bei dem XorShiftHash wird die Zahl mit einer Konstante multipliziert, bitweise verschoben und dann über ein bitweises Exklusiv-Oder mit der nicht-verschobenen Variante kombiniert.

1.2 Aufgabestellung

In der Aufgabe wurde beschrieben, dass mit konstanter Anzahl an Buckets und unter verschiedenen Lastfaktoren die Anzahl an Kollisionen pro Zugriff und „was für sinnvoll gehalten wird“ gemessen werden soll. Für sinnvoll wird gehalten, die Zugriffszeit zu messen, dementsprechend wurde dieses auch getan.

Für sinnvoll wurde ebenfalls gehalten, die Messungen unter anderen Bedingungen zu wiederholen. Da jeder Bucket eines OpenAddressingTables nur 8 Byte groß ist, einer eines CoalescedTables jedoch 24 Byte könnte bei gleichem Speicherverbauch ein 3 mal so großer OpenAddressingTable erstellt werden. Die Messung mit konstanter Anzahl an Buckets benachteiligt stark OpenAddressingTables, deswegen wurde eine zweite Messreihe mit angepassten Größen durchgeführt. Die Herleitung der verwendeten Anzahl an Buckets pro Verfahren ist im Folgenden zu finden.

1.3 Anzahl an Buckets in Abhängigkeit der geforderten Speichergröße

Die Speichergröße wird festgelegt auf die Größe eines OpenAddressingTables mit 2^{15} Buckets, das entspricht 262kB. Für den CoalescedTable sind Buckets 24B groß, also werden $\frac{262kB}{24B} = 10922$ Buckets verwendet.

Für DirectChainingTables ist ein Bucket ebenfalls 8 Byte groß, jedoch werden zusätzlich für jedes Element 16B alloziiert, daher wird vom vorhandanden Speicherlimit zuerst die Anzahl der Elemente multipliziert mit der Größe eines Elements abgezogen und danach die größtmögliche Anzahl an Buckets im Restspeicherplatz alloziiert.

Für SeparateChainingTables ist das Verfahren komplizierter, da bloß Speicherplatz für jedes Element, welches mit einem anderen kollidiert Extra-Platz alloziiert werden muss. Dafür wird dafür ausgegangen, dass grundlegend jedes eingefügte Element 16B Speicherplatz benötigt, jeder leer gebliebene Bucket am Ende des Einfügens jedoch auch 16B. Die Wahrscheinlichkeit, dass ein bestimmter Bucket nach dem Einfügen von n Elementen noch leer ist ist direkt abhängig von der Anzahl der Buckets m und ergibt sich als $p = \left(\frac{m-1}{m}\right)^n$. Der Erwartungswert leerer Buckets ist somit $E = m\left(\frac{m-1}{m}\right)^n$ und der verwendete Platz ist $N = 16B\left(n + m\left(\frac{m-1}{m}\right)^n\right)$. Diese Gleichung wird numerisch für ein möglichst großes m gelöst, für welches $N \leq 262kB$ gilt.

2 Ergebnisse

2.1 Ergebnisse bei Messung nach Aufgabenbedingungen

Name	50%	90%	95%	100%
Quadratic Mul	0,42	1,73	2,35	8,8
Quadratic Mod	0,44	1,72	2,32	10,38
Quadratic XOR	0,43	1,71	2,36	9,54
Linear Mul	0,43	1,87	2,60	8,93
Linear Mod	0,42	1,87	2,64	10,51
Linear XOR	0,44	1,91	2,59	9,83
Triangular Mul	0,42	1,71	2,36	8,85
Triangular Mod	0,43	1,74	2,37	8,12
Triangular XOR	0,43	1,69	2,35	9,71
Direct Mul	0,25	0,44	0,47	0,50
Direct Mod	0,26	0,45	0,48	0,50
Direct XOR	0,25	0,45	0,48	0,51
Separate Mul	0,24	0,44	0,48	0,50
Separate Mod	0,25	0,45	0,47	0,50
Separate XOR	0,25	0,46	0,48	0,51
Coalesced Mul	1,17	1,66	1,74	1,80
Coalesced Mod	1,16	1,66	1,72	1,79
Coalesced XOR	1,17	1,65	1,72	1,79

Tabelle 2.1: Durchschnittliche Kollisionen bei Zugriff mit Erfolg

Name	50%	90%	95%	100%
Quadratic Mul	1,14	9,50	19,51	-
Quadratic Mod	1,14	9,60	19,73	-
Quadratic XOR	1,13	9,62	19,73	-
Linear Mul	1,16	11,14	22,45	-
Linear Mod	1,16	11,01	23,09	-
Linear XOR	1,17	11,14	24,24	-
Triangular Mul	1,12	9,72	19,94	-
Triangular Mod	1,13	9,69	19,68	-
Triangular XOR	1,13	9,74	19,58	-
Direct Mul	0,50	0,90	0,95	0,99
Direct Mod	0,50	0,91	0,94	1,00
Direct XOR	0,50	0,90	0,95	0,99
Separate Mul	0,50	0,90	0,95	0,99
Separate Mod	0,50	0,90	0,96	1,00
Separate XOR	0,50	0,90	0,95	1,00
Coalesced Mul	2,19	2,84	2,97	3,09
Coalesced Mod	2,18	2,84	2,94	3,10
Coalesced XOR	2,18	2,80	2,93	3,06

Tabelle 2.2: Durchschnittliche Kollisionen bei Zugriff mit Erfolg

Name	50%	90%	95%	100%
Quadratic Mul	15,69	22,84	26,38	36,12
Quadratic Mod	7,95	12,98	17,40	26,56
Quadratic XOR	11,11	19,13	13,36	26,82
Linear Mul	22,36	18,60	18,47	28,08
Linear Mod	7,07	11,77	12,42	21,82
Linear XOR	12,34	16,40	13,42	42,18
Triangular Mul	12,73	22,71	19,54	44,04
Triangular Mod	6,40	11,02	11,95	20,09
Triangular XOR	8,25	14,00	13,82	27,00
Direct Mul	29,27	18,00	18,34	18,84
Direct Mod	11,03	30,81	15,37	15,18
Direct XOR	15,89	16,96	17,24	19,36
Separate Mul	11,61	15,89	22,11	24,84
Separate Mod	8,80	12,73	25,38	13,32
Separate XOR	10,45	14,45	14,89	21,30
Coalesced Mul	13,39	22,21	18,46	18,33
Coalesced Mod	10,66	15,11	16,11	15,69
Coalesced XOR	12,40	23,61	16,96	18,37

Tabelle 2.3: Durchschnittliche Kollisionen bei Zugriff mit Erfolg

Name	50%	90%	95%	100%
Quadratic Mul	26,70	54,74	63,56	-
Quadratic Mod	29,43	34,32	47,58	-
Quadratic XOR	23,65	33,29	42,50	-
Linear Mul	28,92	43,14	49,96	-
Linear Mod	17,75	34,40	41,58	-
Linear XOR	21,51	32,34	45,58	-
Triangular Mul	24,55	45,98	51,65	-
Triangular Mod	16,08	29,74	42,57	-
Triangular XOR	19,55	33,58	45,38	-
Direct Mul	31,02	30,57	31,61	31,03
Direct Mod	19,57	38,78	26,45	26,70
Direct XOR	23,20	44,13	29,42	30,70
Separate Mul	23,49	28,22	30,52	31,97
Separate Mod	27,56	23,86	34,34	24,72
Separate XOR	22,51	27,11	27,73	29,54
Coalesced Mul	18,64	33,36	31,12	32,21
Coalesced Mod	15,82	25,53	32,82	29,31
Coalesced XOR	17,42	34,61	30,27	32,26

Tabelle 2.4: Durchschnittliche Kollisionen bei Zugriff mit Erfolg

2.2 Ergebnisse bei Messung nach für sinnvoll gehaltenen Bedingungen

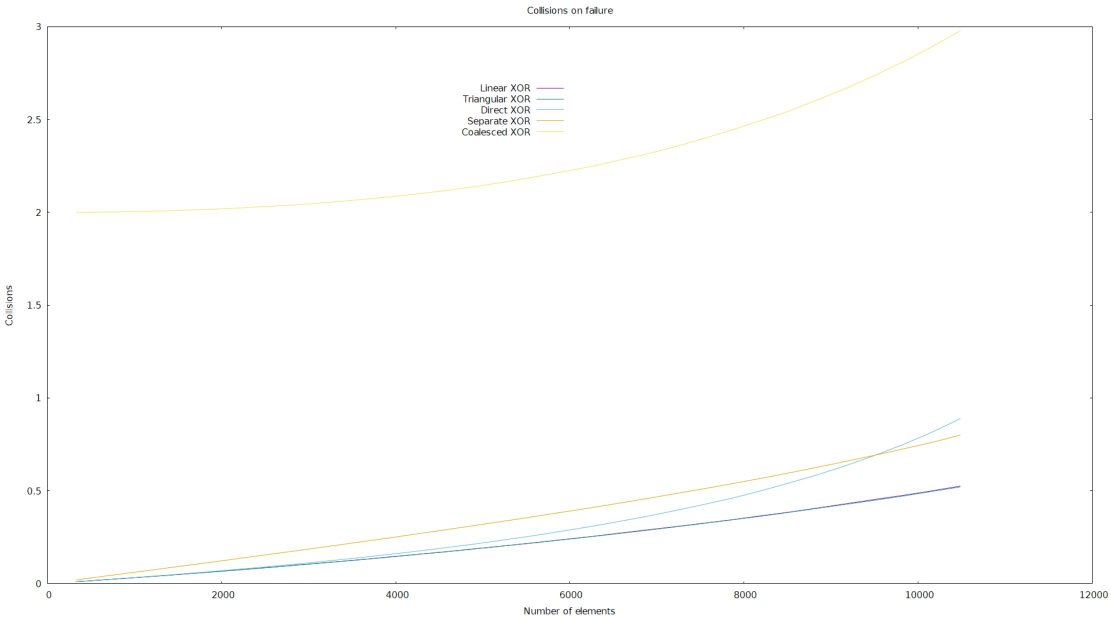

Abbildung 2.1: Durchschnittliche Anzahl an Kollisionen bei fehlschlagenden Zugriffen

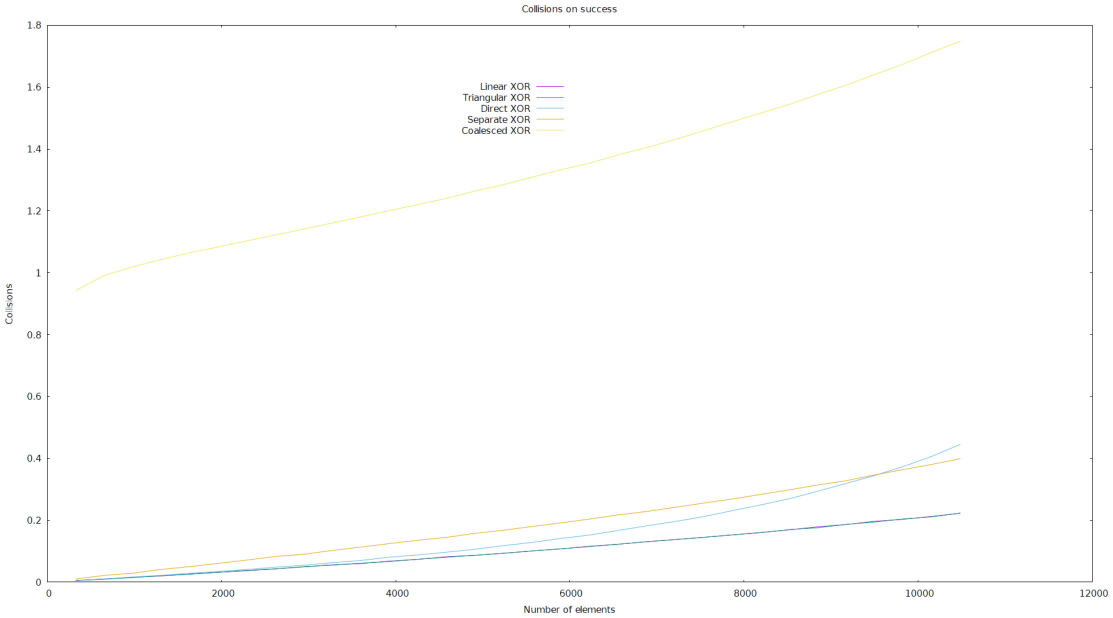

Abbildung 2.2: Durchschnittliche Anzahl an Kollisionen bei erfolgreichen Zugriffen

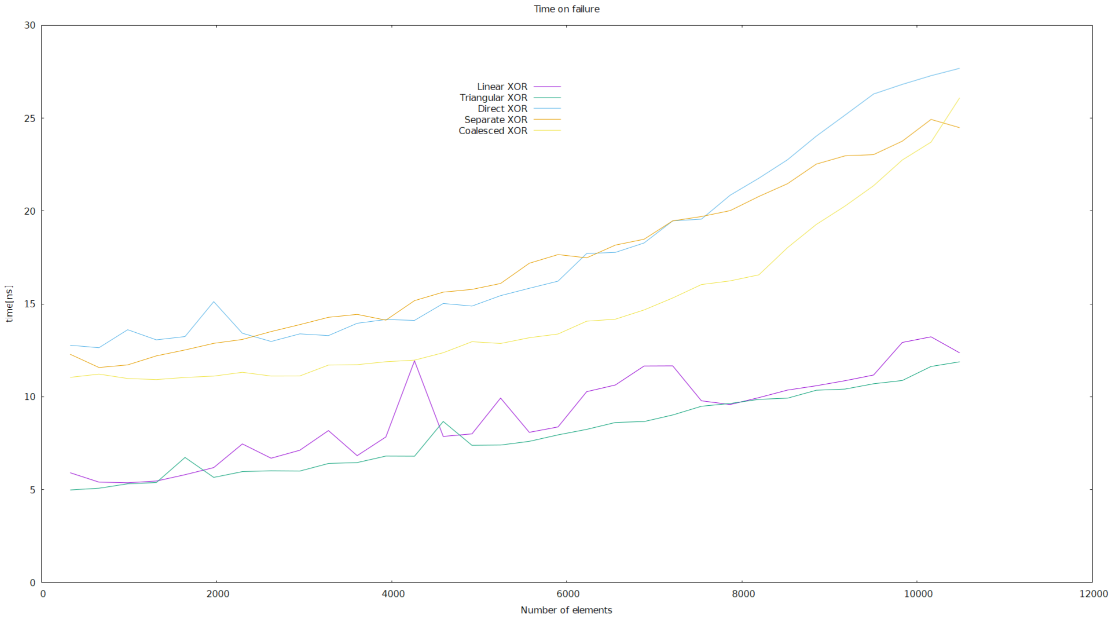

Abbildung 2.3: Durchschnittliche Zugriffszeit bei fehlschlagenden Zugriffen in ns

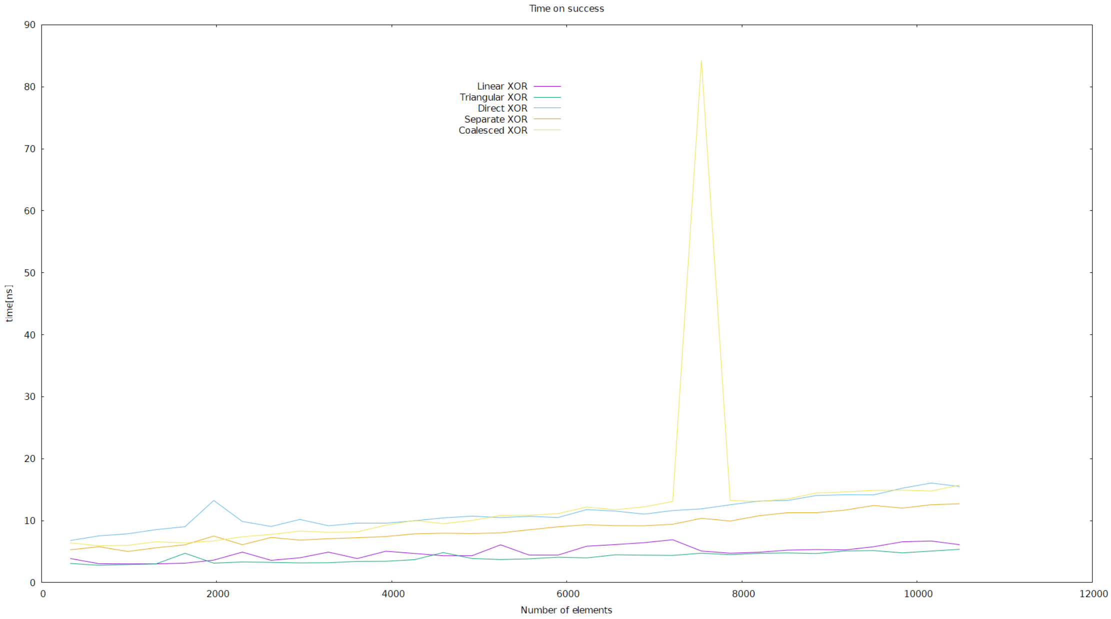

Abbildung 2.4: Durchschnittliche Zugriffszeit bei erfolgreichen Zugriffen in ns

3 Interpretation der Ergebnisse

3.1 Interpretation der Egebnisse unter den Bedingungen der Aufgabenstellung

3.2 Interpration der Ergebnisse unter für sinnvoll gehaltenen Bedingungen

4 Programm- bzw. Quellcode

4.1 Quellcode

Der komplette Quellcode inklusive Graph-Generierung wurde in Rust geschrieben. Die Ausführung sollte über das Tool „cargo“ durchgeführt werden. Das erhaltene Executable gibt die gemessenen Daten in Tabellenform in stdout aus, erstellt eine csv-Datei mit den Daten und erstellt 4 Graphen. Die Konfiguration der Messung erfolgt über den Quellcode.

Da gefordert wurde, dass der komplette Quellcode im Dokument enthalten ist, hier eine Sektion mit demselben. Alternativ kann das Projekt auch angenehm auf GitHub unter [1] eingesehen werden. Die Dokumentation und Erklärung des Codes ist über „Doc Comments“ realisiert, kann also im Code durch Kommentare komplett eingebettet gelesen werden. Alternativ kann auch über den Aufruf des Befehls `cargo doc` eine HTML-Dokumentation generiert werden.

4.2 Vollständiger Quellcode

Listing 4.1: main.rs

```
//! Generates statistics for different types of tables
//!
//! Use by changing the constants found throughout the program
//! Variants tested can be adjusted by changing the vec ‘tables‘ in main
//! load factors tested can be adjusted by changing the constant ‘
    ↪ LOAD_FACTORS‘

extern crate gnuplot;
extern crate rand;

```

[1] https://github.com/imkgerC/uni-theo2-hashset

```
pub mod hashset;
pub mod logging;

use hashset::*;
use logging::*;
use rand::{thread_rng, Rng};
use std::time::Instant;

/// Helper function to get an instance of a DefaultHashTableBuilder for
    ↪ the given HashTable
fn get_builder<T: PartialEq + 'static, H: 'static + HashTable<T> +
    ↪ Default>(
) -> Box<dyn HashTableBuilder<T>> {
    Box::new(DefaultHashTableBuilder::<T, H>::default())
}

/// Resizes every type of HashTable, so they take up
/// nearly the same space in memory
const RESIZE_TO_MAKE_FAIR: bool = true;
/// How many elements to insert into the HashTable before doing
/// probing tests
const LOAD_FACTORS: [f64; 32] = [
    0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11,
    ↪ 0.12, 0.13, 0.14, 0.15, 0.16,
    0.17, 0.18, 0.19, 0.2, 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27,
    ↪ 0.28, 0.29, 0.3, 0.31, 0.32,
];
/// How many tests to do at each load factor
const ITERATIONS_PER_LOAD_FACTOR: usize = 50;

fn main() {
    // All variants of HashTable possible in this module
    #[rustfmt::skip]
    let tables: Vec<(Box<dyn HashTableBuilder<u32>>, String)> = vec![
     (get_builder::<u32, OpenAddressingTable::<u32, QuadraticProber,
↪ MulHash>>(), "Quadratic␣Mul".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, QuadraticProber,
↪ ModHash>>(), "Quadratic␣Mod".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, QuadraticProber,
↪ XorShiftHash>>(), "Quadratic␣XOR".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, LinearProber,
↪ MulHash>>(), "Linear␣Mul".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, LinearProber,
↪ ModHash>>(), "Linear␣Mod".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, LinearProber,
↪ XorShiftHash>>(), "Linear␣XOR".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, TriangularProber,
↪  MulHash>>(), "Triangular␣Mul".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, TriangularProber,
↪  ModHash>>(), "Triangular␣Mod".to_owned()),
     (get_builder::<u32, OpenAddressingTable::<u32, TriangularProber,
↪  XorShiftHash>>(), "Triangular␣XOR".to_owned()),
     (get_builder::<u32, DirectChainingTable::<u32, MulHash>>(), "
↪ Direct␣Mul".to_owned()),
     (get_builder::<u32, DirectChainingTable::<u32, ModHash>>(), "
↪ Direct␣Mod".to_owned()),
     (get_builder::<u32, DirectChainingTable::<u32, XorShiftHash>>(),
↪  "Direct␣XOR".to_owned()),
     (get_builder::<u32, SeparateChainingTable::<u32, MulHash>>(), "
↪ Separate␣Mul".to_owned()),
     (get_builder::<u32, SeparateChainingTable::<u32, ModHash>>(), "
↪ Separate␣Mod".to_owned()),
     (get_builder::<u32, SeparateChainingTable::<u32, XorShiftHash
↪ >>(), "Separate␣XOR".to_owned()),
     (get_builder::<u32, CoalescedTable::<u32, MulHash>>(), "
↪ Coalesced␣Mul".to_owned()),
     (get_builder::<u32, CoalescedTable::<u32, ModHash>>(), "
↪ Coalesced␣Mod".to_owned()),
        (get_builder::<u32, CoalescedTable::<u32, XorShiftHash>>(), "
    ↪ Coalesced␣XOR".to_owned()),
    ];
    generate_stats(tables);
}

/// generates and outputs stats
///
/// generates and outputs stats for every table in ‘tables‘.
/// Stats are: How many collisions on successful find, how many
    ↪ collisions on
/// failed find, how much time on successful find, how much time on
    ↪ failed find;
/// Stats are output to stdout, a hardcoded csv file and 4 graphs (one
    ↪ for every stat)
/// Stats are calculated at every load_factor in LOAD_FACTORS
/// ITERATIONS_PER_LOAD_FACTOR times
fn generate_stats(tables: Vec<(Box<dyn HashTableBuilder<u32>>, String)>)
    ↪  {
    let mut all_stats = Vec::new();

    for (builder, name) in tables {
        let mut stats = Vec::new();
        for s in &LOAD_FACTORS {
            let mut stats_for_this = (0_f32, 0_f64, 0_f32, 0_f64);
            for _ in 0..ITERATIONS_PER_LOAD_FACTOR {
                let temp = get_stats(builder.as_ref(), *s);
                stats_for_this.0 += temp.0;
                stats_for_this.1 += temp.1;
                stats_for_this.2 += temp.2;
                stats_for_this.3 += temp.3;
            }
            stats_for_this.0 /= ITERATIONS_PER_LOAD_FACTOR as f32;
            stats_for_this.1 /= ITERATIONS_PER_LOAD_FACTOR as f64;
            stats_for_this.2 /= ITERATIONS_PER_LOAD_FACTOR as f32;
            stats_for_this.3 /= ITERATIONS_PER_LOAD_FACTOR as f64;
            stats.push(stats_for_this);
        }
        // print stats for this table
        print_subtable(&name, &stats, &LOAD_FACTORS);
        all_stats.push((name, stats));
    }

    // create output file for analysis in csv format
    write_csv(&all_stats, &LOAD_FACTORS);

    // create graph for every type of HashTable
    write_graphs(&all_stats, &LOAD_FACTORS, ELEMENT_COUNT);
}

/// get stats for one type of hash table
///
/// fills the HashTable with ‘fill‘ values and then takes measurements
/// for different statistics
fn get_stats(builder: &dyn HashTableBuilder<u32>, fill: f64) -> (f32,
   ↪ f64, f32, f64) {
    let fill = f64::min(fill * ELEMENT_COUNT as f64, ELEMENT_COUNT as
   ↪ f64) as usize;
    get_stats_rec(builder, fill, 0)
}

/// recursively tries to get stats
///
/// fills the HashTable with ‘fill‘ values and then takes measurements
/// for different statistics. If it fails at any point it tries again.
/// One reason for failure could be a nearly full OpenAddressingTable
/// with QuadraticProbing. At most, 100 attempts are allowed
fn get_stats_rec(
    builder: &dyn HashTableBuilder<u32>,
    fill: usize,
    attempt: usize,
) -> (f32, f64, f32, f64) {
    // amount of samples to test at random
    let random_samples = 1_usize << 16;

    let mut table = builder.build();
    // resize if needed
    if RESIZE_TO_MAKE_FAIR {
        table.as_mut().resize_to_bytes(ELEMENT_COUNT << 3, fill);
    }

    // fill hash set with ‘fill‘ random values
    let mut rng = thread_rng();
    let mut inserted_nums = Vec::with_capacity(fill);
    for _ in 0..fill {
        let num = rng.gen();
        inserted_nums.push(num);
        if !HashTable::insert(table.as_mut(), &num) {
            // try again, up to 100 times
            if attempt > 100 {
                return (std::f32::NAN, std::f64::NAN, std::f32::NAN, std
   ↪ ::f64::NAN);
            }
            return get_stats_rec(builder, fill, attempt + 1);
        }
    }
    let mut ns = 0_usize; // number of successful reads
    let mut nf = 0_usize; // number of failed reads
    let mut cs = 0_usize; // collisions on successful reads
    let mut cf = 0_usize; // collisions on failed reads
    let start_time = Instant::now();
    for x in &inserted_nums {
        table.as_mut().has(x);
    }
    // duration of ‘fill‘ successful reads
    let duration_s = start_time.elapsed().as_nanos();
    let start_time = Instant::now();
    for _ in 0..random_samples {
        let num = rng.gen();
        table.as_mut().has(&num);
    }
    // we assume random numbers nearly always fail
    // therefore: duration of ‘random_samples‘ failed reads
    let duration_f = start_time.elapsed().as_nanos();

    // First try all numbers we already inserted, so we guarantee
    // some successful reads
    for x in &inserted_nums {
        HashTable::reset_collisions(table.as_mut());
        if HashTable::has(table.as_mut(), x) {
            ns += 1;
            cs += HashTable::get_collisions(table.as_ref());
        } else {
            println!("did␣not␣find␣what␣we␣would␣need␣to␣find");
            nf += 1;
            cf += HashTable::get_collisions(table.as_ref());
        }
    }
    // Then always try 2^16 more reads with random samples
    for _ in 0..(1_usize << 16) {
        let num = rng.gen();
        HashTable::reset_collisions(table.as_mut());
        if HashTable::has(table.as_mut(), &num) {
            ns += 1;
            cs += HashTable::get_collisions(table.as_ref());
        } else {
            nf += 1;
            cf += HashTable::get_collisions(table.as_ref());
        }
    }
    let nf = nf as f32;
    let cf = cf as f32;
    let ns = ns as f32;
    let cs = cs as f32;
    (
        (cs / ns), // average number of collisions on success
        (duration_s as f64 / fill as f64), // average time on success
        (cf / nf), // average number of collisions on fail
        (duration_f as f64 / random_samples as f64), // average time on fail
    )
}
```

Listing 4.2: mod.rs

```
//! Module containing everything relevant to hashsets
//!
//! This contains hashing functions, probing functions,
//! HashTable implementations and HashTable builders

mod chainingtable;
mod coalescedtable;
mod hashing;
mod openaddressing;
mod probing;

pub use chainingtable::*;
pub use coalescedtable::*;
pub use hashing::*;
pub use openaddressing::*;
pub use probing::*;
```

```
use std::marker::PhantomData;

/// Number of buckets for the OpenAddressingTable and load_factor is
  ↪ based on
pub const ELEMENT_COUNT: usize = 1 << 15;

/// Generic HashTable as set datastructure
///
/// The hashtable should only count collisions on calls for finding an
  ↪ element
/// not on insertion. Every inserted element should only be saved once,
/// as only either having or not having the element is checked.
/// The HashTable can not delete any entries or dynamically resize the
  ↪ table
pub trait HashTable<T> {
    /// checks if the element is in the set
    ///
    /// Should return false if the element can not be found and true if
  ↪ it can
    /// Needs to count every collision that occured during the check
    fn has(&mut self, val: &T) -> bool;
    /// resets collisions
    fn reset_collisions(&mut self);
    /// returns the number of collisions
    fn get_collisions(&self) -> usize;
    /// inserts the element in the HashTable
    ///
    /// returns true if the element is already in the HashTable
    /// returns true if the element was successfully inserted
    /// returns false iff the element cannot be inserted into the
  ↪ HashTable
    ///
    /// ## Causes for failure
    /// - the hashset is full
    /// - a bad cycle in probing hindered insertion
    fn insert(&mut self, val: &T) -> bool;
    /// resize the number of buckets to most closely match the number of
   ↪  bytes used
    ///
    /// depending on the type of HashTable it could be hard to implement
   ↪  with
    /// 100% accuracy. Therefore the expected value of the number
    /// of bytes can be used as an approximate value
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize);
}

/// A generic builder for HashTables
///
/// The trait can be used when a generic HashTable type is to be
/// used as a function parameter
pub trait HashTableBuilder<T> {
    /// returns an instance of HashTable
    fn build(&self) -> Box<dyn HashTable<T>>;
}

/// Default implementation of HashTableBuilder
///
/// Just calls the Default trait of the inner HashTable type.
/// Is the most basic implementation of a HashTableBuilder
/// Should be the most used one
pub struct DefaultHashTableBuilder<T: PartialEq, H: HashTable<T> +
   ↪ Default> {
    // PhantomData is used to not actually save any data
    // It is only used to save the type to be instantiated
    table: PhantomData<H>,
    t: PhantomData<T>,
}
```

```
impl<T: PartialEq, H: 'static + HashTable<T> + Default> HashTableBuilder
   ↪ <T>
    for DefaultHashTableBuilder<T, H>
{
    /// returns the default instance of HashTable
    fn build(&self) -> Box<dyn HashTable<T>> {
        Box::new(H::default())
    }
}

impl<T: PartialEq, H: HashTable<T> + Default> Default for
   ↪ DefaultHashTableBuilder<T, H> {
    fn default() -> Self {
        Self {
            table: PhantomData,
            t: PhantomData,
        }
    }
}
```

Listing 4.3: hashing.rs

```
/// Simple modulo hasher
///
/// This is the most simple hashing function one could fathom.
/// It does not calculate modulo a big prime but only modulo the maximum
pub struct ModHash;
impl Hasher<u32> for ModHash {
    fn hash(val: &u32, max: usize) -> usize {
        *val as usize % max
    }
}

/// Theoretically nice multiplicative hasher
///
```

```
/// This works by multiplying with the golden ratio, which is the number hardest to
/// approximate with fractions. This makes for a pseudo-random decimal fraction part
/// which is used as the base for the hash. This is then multiplied with the maximum
pub struct MulHash;
const PHI: f64 = 0.618_033_988_75;
impl Hasher<u32> for MulHash {
    fn hash(val: &u32, max: usize) -> usize {
        let val = *val as f64;
        (max as f64 * ((val * PHI) - f64::floor(val * PHI))) as usize
    }
}

/// XOR shift hasher
///
/// This works by using the value as the seed for a pseudo-random XORShiftRng
pub struct XorShiftHash;
impl Hasher<u32> for XorShiftHash {
    fn hash(val: &u32, max: usize) -> usize {
        let x = *val;
        let x = ((x >> 16) ^ x).wrapping_mul(0x45d_9f3b_u32);
        let x = ((x >> 16) ^ x).wrapping_mul(0x45d_9f3b_u32);
        let x = (x >> 16) ^ x;
        x as usize % max
    }
}

/// Minimal trait for hashing functions
///
/// Every hasher is a non-instantiable struct with a static method for hashing
```

```rust
/// The hashing trait is implemented for a specific type
pub trait Hasher<T> {
    /// Hashing function
    ///
    /// - val: A reference to the value to hash
    /// - max: The length of the hashset
    /// returns: An integer value in the interval [0, max)
    fn hash(val: &T, max: usize) -> usize;
}
```

Listing 4.4: probing.rs

```rust
/// Trait for Prober for OpenAdressingTables
///
/// The Prober provides the offset to add at the ith attempt
pub trait Prober {
    /// Provides the offset at the ith attempt
    ///
    /// The implementor should provide an offset. There are no
   ↪ limitations on the range of the offset
    fn probe(i: usize) -> usize;
}

/// Triangular Probing
///
/// The triangular prober always has the sum from 0 to i as the offset
/// It uses the gaussian formula for summation. This approach is all
   ↪ used in
/// real world high-performance implementations
pub struct TriangularProber;
impl Prober for TriangularProber {
    fn probe(i: usize) -> usize {
        (i * (i + 1)) >> 1 // sum (0, .., i) = (i(i+1))/2
    }
}
```

```

/// Simplest prober
///
/// The offset is i. It tries all buckets in linear fashion
pub struct LinearProber;
impl Prober for LinearProber {
    fn probe(i: usize) -> usize {
        i
    }
}

/// Quadratic probing is not used in pracitcal applications anymore
///
/// The offset is simple i*i. This makes for an easy implementation with less clustering than linear probing.
/// An issue with this method is that not every value forms a cycle over every bucket, so the table may become full before
/// every bucket is used.
pub struct QuadraticProber;
impl Prober for QuadraticProber {
    fn probe(i: usize) -> usize {
        i * i
    }
}
```

Listing 4.5: chainingtable.rs

```
use super::{HashTable, Hasher, ELEMENT_COUNT};
use std::marker::PhantomData;
use std::mem::size_of;

/// Simple singly-linked List
///
/// This implementation is not optimized for insertion performance
/// but for size. The implementation is naive and should not be used
/// for anything serious.
pub enum LinkedList<T> {
    Cons(T, Box<LinkedList<T>>),
    Nil,
}

impl<T: PartialEq + Copy> LinkedList<T> {
    /// checks if an element is contained in the list
    ///
    /// searches the LinkedList recursively with the canonical approach
    pub fn contains(&self, searched: &T) -> bool {
        match self {
            LinkedList::Cons(val, other) => *val == *searched || other.
   ↪ contains(searched),
            LinkedList::Nil => false,
        }
    }

    /// inserts the Value as a new node at the end of the list
    ///
    /// This function is a really slow implementation. It was only
    /// implemented this way because of simplicity
    pub fn push(&mut self, val: T) {
        let push_to_self = match self {
            LinkedList::Cons(_, other) => {
                other.push(val);
                false
            }
            LinkedList::Nil => true,
        };
        if push_to_self {
            *self = LinkedList::Cons(val, Box::new(LinkedList::Nil));
        }
    }
}

/// Direct chaining implementation of HashTable
///
/// Every bucket is a pointer to a LinkedList that is used for
/// collision resolution. An infinite amount of elements can be
/// inserted into this table
pub struct DirectChainingTable<T: PartialEq + Copy, H: Hasher<T>> {
    collisions: usize,
    entries: Vec<Box<LinkedList<T>>>,
    hasher: PhantomData<H>,
}
impl<T: PartialEq + Copy, H: Hasher<T>> Default for DirectChainingTable<
   ↪ T, H> {
    /// initializes HashTable with ELEMENT_COUNT buckets
    fn default() -> Self {
        Self::with_size(ELEMENT_COUNT)
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> DirectChainingTable<T, H> {
    /// initializes HashTable with ‘size‘ buckets
    fn with_size(size: usize) -> Self {
        let mut entries = Vec::with_capacity(size);
        for _ in 0..size {
            entries.push(Box::new(LinkedList::Nil));
        }
        Self {
            collisions: 0,
            entries,
            hasher: PhantomData,
        }
    }
}
```

```
impl<T: PartialEq + Copy, H: Hasher<T>> HashTable<T> for
   ↪ DirectChainingTable<T, H> {
    /// checks table for value
    ///
    /// checks by checking the LinkedList at the correct bucket.
    /// Counts the number of collisions
    fn has(&mut self, val: &T) -> bool {
        let index = H::hash(val, self.entries.len());
        let mut right_now = self.entries[index].as_ref();
        while let LinkedList::Cons(x, next) = right_now {
            if *x == *val {
                return true;
            }
            self.collisions += 1;
            right_now = next.as_ref();
        }
        false
    }
    /// resets number of collisions
    fn reset_collisions(&mut self) {
        self.collisions = 0;
    }
    /// returns number of collisions
    fn get_collisions(&self) -> usize {
        self.collisions
    }
    /// inserts the element into the HashTable
    ///
    /// always returns true as it won't fail
    fn insert(&mut self, val: &T) -> bool {
        let index = H::hash(val, self.entries.len());
        if !self.entries[index].contains(val) {
            self.entries[index].push(*val);
        }
        true
    }

    /// resizes the number of buckets to specified byte value
    ///
    /// size of a direct chaining table is
    /// buckets*(size of a bucket) + entries*(size of a node)
    /// (for T = u32) size of a bucket is the size of a
    /// pointer to a LinkedList = 8
    /// size of a node is the size of a LinkedList node = 16
    /// Fails if bytes - 16*elements < 8
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize) {
        let list_size = size_of::<LinkedList<T>>();
        let available_bytes = bytes as isize - (list_size * elements) as
   ↪  isize;
        if available_bytes < size_of::<Box<LinkedList<T>>>() as isize {
            panic!("not␣enough␣bytes␣available␣for␣the␣buckets");
        }
        *self = Self::with_size(available_bytes as usize / size_of::<Box
   ↪ <LinkedList<T>>>());
    }
}

/// Separate chaining implementation of HashTable
///
/// Every bucket has stores one value and a pointer to a LinkedList
/// that is used for collision resolution. An infinite amount of
/// elements can be inserted into this table.
pub struct SeparateChainingTable<T: PartialEq + Copy, H: Hasher<T>> {
    collisions: usize,
    entries: Vec<(Option<T>, Box<LinkedList<T>>)>,
    hasher: PhantomData<H>,
}
```

```
impl<T: PartialEq + Copy, H: Hasher<T>> Default for
   ↪ SeparateChainingTable<T, H> {
    /// initializes HashTable with ELEMENT_COUNT buckets
    fn default() -> Self {
        Self::with_size(ELEMENT_COUNT)
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> SeparateChainingTable<T, H> {
    /// initializes HashTable with ‘size‘ buckets
    fn with_size(size: usize) -> Self {
        let mut entries = Vec::with_capacity(size);
        for _ in 0..size {
            entries.push((None, Box::new(LinkedList::Nil)));
        }
        Self {
            collisions: 0,
            entries,
            hasher: PhantomData,
        }
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> HashTable<T> for
   ↪ SeparateChainingTable<T, H> {
    /// checks table for value
    ///
    /// checks by checking first checking the value stored at the
    /// correct bucket, then checking the associated LinkedList.
    /// Counts the number of collisions
    fn has(&mut self, val: &T) -> bool {
        let index = H::hash(val, self.entries.len());
        if let Some(x) = self.entries[index].0 {
            if x == *val {
                return true;
            }
            self.collisions += 1;
        }
        let mut right_now = self.entries[index].1.as_ref();
        while let LinkedList::Cons(x, new) = right_now {
            if *x == *val {
                return true;
            }
            self.collisions += 1;
            right_now = new.as_ref();
        }
        false
    }
    /// resets number of collisions
    fn reset_collisions(&mut self) {
        self.collisions = 0;
    }
    /// returns number of collisions
    fn get_collisions(&self) -> usize {
        self.collisions
    }
    /// inserts the element into the HashTable
    ///
    /// always returns true as it won’t fail
    fn insert(&mut self, val: &T) -> bool {
        let index = H::hash(val, self.entries.len());
        if self.entries[index].0.is_none() {
            self.entries[index].0 = Some(*val);
            return true;
        }
        if let Some(x) = self.entries[index].0 {
            if x == *val {
                return true;
            }
        }
        if self.entries[index].1.contains(val) {
            return true;
        }
        self.entries[index].1.push(*val);
        true
    }
    /// resizes the number of buckets to specified byte value
    ///
    /// Warning: Is correct for T=u32 only.
    /// (for T=u32) The size of a separate chaining table is
    /// elements*16 + empty_buckets*16
    /// The number of empty buckets is approximated with m((m-1)/m)^n
    /// As this equation grows monotonically in m, the value can be
   ↪ found
    /// by using a variant of binary search.
    /// Will fail if no suitable value m is found.
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize) {
        // to make calculations simpler we do not calculate in terms of
        // bytes but in terms of 16B always
        let available_elements = (bytes / 16) as f64;

        let elements = elements as f64;
        // We cannot allocate more buckets than with zero collisions
        let mut buckets = available_elements as f64;
        let mut step = available_elements as f64 / 2_f64;
        // do to an accuracy of 1
        while step > 1_f64 {
            let used_elements = buckets * ((buckets - 1_f64) / buckets).
   ↪ powf(elements) + elements;
            if used_elements > available_elements {
                // found a maximum, go lower
                buckets -= step;
            } else if used_elements < available_elements {
                // found a minimum, can lower step size
                step = step / 2_f64;
                buckets += step;
            }
        }
        if buckets < 1_f64 {
            panic!("invalid␣configuration␣for␣direct␣chaining␣table");
        }
        *self = Self::with_size(buckets as usize);
    }
}
```

Listing 4.6: openaddressing.rs

```
use super::{HashTable, Hasher, Prober, ELEMENT_COUNT};
use std::marker::PhantomData;

/// Simple and fast HashTable with OpenAddressing
///
/// OpenAddressing is used for collision resolution. The number of
/// buckets is always equal to ELEMENT_COUNT. If quadratic probing is
/// used, insertion could fail even though not every bucket is used.
pub struct OpenAddressingTable<T: PartialEq + Copy, P: Prober, H: Hasher
    ↪ <T>> {
    collisions: usize,
    entries: [Option<T>; ELEMENT_COUNT],
    prober: PhantomData<P>,
    hasher: PhantomData<H>,
}

impl<T: PartialEq + Copy, P: Prober, H: Hasher<T>> Default for
    ↪ OpenAddressingTable<T, P, H> {
    fn default() -> Self {
        Self {
            collisions: 0,
            entries: [None; ELEMENT_COUNT],
            prober: PhantomData,
            hasher: PhantomData,
        }
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>, P: Prober> HashTable<T> for
   ↪ OpenAddressingTable<T, P, H> {
    /// probes table for value
    ///
    /// returns true iff value was inserted into HashTable
    /// It will at maximum check a number of buckets equal to the
    /// total number of buckets. It does not use cycle detection.
    /// While probing the hash+offset is wrapped around the end of the
   ↪ table.
    /// Every accessed non-empty bucket that did not contain the value
    /// searched for is counted as a collision
    fn has(&mut self, val: &T) -> bool {
        let mut index = H::hash(val, self.entries.len());
        let mut attempts = 0;
        while attempts < self.entries.len() {
            if let Some(inside) = self.entries[index] {
                if inside == *val {
                    return true;
                }
            } else {
                return false;
            }
            attempts += 1;
            self.collisions += 1;
            index = (index + P::probe(attempts)) % self.entries.len();
        }
        false
    }
    /// resets number of collisions
    fn reset_collisions(&mut self) {
        self.collisions = 0;
    }
    /// returns number of collisions
    fn get_collisions(&self) -> usize {
        self.collisions
    }
    /// inserts the element in the HashTable if possible
    ///
    /// returns true iff the value was inserted successfully
    /// It will at maximum check a number of buckets equal to the
    /// total number of buckets. It does not use cycle detection.
    /// Insertion is not optimized for performance.
    /// Insertion does not count collisions.
    fn insert(&mut self, val: &T) -> bool {
        if self.has(val) {
            return true;
        }
        let mut index = H::hash(val, self.entries.len());
        let mut attempts = 0;
        while attempts < self.entries.len() {
            if self.entries[index].is_none() {
                self.entries[index] = Some(*val);
                return true;
            }
            attempts += 1;
            index = (index + P::probe(attempts)) % self.entries.len();
        }
        false
    }
    /// fails if bytes unequal to 8*ELEMENT_COUNT (for T = u32)
    ///
    /// As the OpenAddressingTable is backed by an array with static
   ↪ size,
    /// it can't be dynamically allocated or resized. Thusly this
    /// method will fail if not called with bytes =
    /// ELEMENT_COUNT*size_of(Option<T>) and elements <= ELEMENT_COUNT
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize) {
        if elements > ELEMENT_COUNT {
            panic!("trying␣to␣insert␣more␣elements␣than␣possible␣by␣
   ↪ constraint");
        }
        if bytes * std::mem::size_of::<Option<T>>() != ELEMENT_COUNT {
            panic!("trying␣to␣resize␣to␣invalid␣size");
        }
    }
}
```

Listing 4.7: coalescedtable.rs

```
use super::{HashTable, Hasher, ELEMENT_COUNT};
use std::marker::PhantomData;

/// HashTable with coalesced buckets for collision resolution
///
/// Every buckets saves an Element and an optional pointer to
/// the next bucket used for collision resolution.
pub struct CoalescedTable<T: PartialEq + Copy, H: Hasher<T>> {
    collisions: usize,
    entries: Vec<Option<(T, Option<usize>)>>,
    hasher: PhantomData<H>,
    cursor: usize,
}
impl<T: PartialEq + Copy, H: Hasher<T>> Default for CoalescedTable<T, H>
   ↪  {
    /// initializes HashTable with ELEMENT_COUNT buckets
    fn default() -> Self {
        Self::with_size(ELEMENT_COUNT)
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> CoalescedTable<T, H> {
    /// initializes HashTable with ‘size‘ buckets
    fn with_size(size: usize) -> Self {
        let mut entries = Vec::with_capacity(size);
        for _ in 0..size {
            entries.push(None);
        }
        Self {
            collisions: 0,
            entries,
            hasher: PhantomData,
            cursor: 0,
        }
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> HashTable<T> for CoalescedTable<
   ↪ T, H> {
    /// checks table for value
    ///
    /// checks the table through the efficient algorithm used in
    /// separate chaining.
    fn has(&mut self, val: &T) -> bool {
        let mut index = H::hash(val, self.entries.len());
        if self.entries[index].is_none() {
            self.entries[index] = Some((*val, None));
            return true;
        }
        self.collisions += 1;
        loop {
            if let Some((x, next)) = self.entries[index] {
                if x == *val {
                    return true;
                }
                self.collisions += 1;
                if let Some(i) = next {
                    index = i;
                } else {
                    break;
                }
            } else {
                panic!("data␣inconsistency");
            }
        }
        false
    }
    /// resets number of collisions
    fn reset_collisions(&mut self) {
        self.collisions = 0;
    }
    /// returns number of collisions
    fn get_collisions(&self) -> usize {
        self.collisions
    }

    /// inserts an element into the table
    ///
    /// returns true iff the value was inserted successfully
    /// Only fails iff the table is full and the value was
    /// not inserted already
    /// Insertion is not optimized for performance.
    /// Insertion does not count collisions.
    fn insert(&mut self, val: &T) -> bool {
        let mut index = H::hash(val, self.entries.len());
        if self.entries[index].is_none() {
            self.entries[index] = Some((*val, None));
            return true;
        }
        loop {
            if let Some((x, next)) = self.entries[index] {
                if x == *val {
                    return true;
                }
                if let Some(i) = next {
                    index = i;
                } else {
                    break;
                }
            } else {
                panic!("data␣inconsistency");
            }
        }
        while self.cursor < self.entries.len() {
            if self.entries[self.cursor].is_none() {
                self.entries[self.cursor] = Some((*val, None));
                let old = self.entries[index].expect("data␣inconsistency
   ↪ ").0;
                self.entries[index] = Some((old, Some(self.cursor)));
                return true;
            }
            self.cursor += 1;
        }
        true
    }
    /// resizes the number of buckets to specified byte value
    ///
    /// (for T = u32) Every bucket has a size of 24B. If elements*24 >
   ↪ bytes
    /// this method will fail. Otherwise it resizes the hashtable to
    /// bytes / 24 buckets.
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize) {
        let entries = bytes / std::mem::size_of::<Option<(T, Option<
   ↪ usize>)>>();
        if entries < elements {
            panic!("cannot␣resize␣that␣low");
        }
        *self = Self::with_size(entries);
    }
}
```

Listing 4.8: logging.rs

```
//! Module for helper logging functions
//!
//! This module contains all functions for writing output
use gnuplot::{AxesCommon, Caption, Figure, Graph};
use std::fs::OpenOptions;
use std::io::Write;

/// Print a simple header for the table
///
/// # Example
/// Name      | 50% | 90% | 95% | 100%
pub fn print_header(name: &str, load_factors: &[f64]) {
    let mut out = format!("{:20}", name);
    for (i, lambda) in load_factors.iter().enumerate() {
        let lambda = format!("{:.0}%", lambda * 100_f64);
        out.push_str(&format!("{:^5}", lambda));
        if i != load_factors.len() - 1 {
            out.push_str("|");
        }
    }
    println!("{}", out);
}

/// Prints a table inclusive header
///
/// # Example
/// Name                | 50% | 90% | 95% | 100%
/// + collisions        | val | val | val | val
/// + time[ns]          | val | val | val | val
/// - collisions        | val | val | val | val
/// - time[ns]          | val | val | val | val
pub fn print_subtable(name: &str, stats: &[(f32, f64, f32, f64)],
   ↪ load_factors: &[f64]) {
    println!();
    print_header(name, load_factors);
    let mut out = format!("{:20}", "+␣collisions");
    for i in 0..stats.len() {
        out.push_str(&format!("{:^5.2}", stats[i].0));
        if i != stats.len() - 1 {
            out.push_str("|");
        }
    }
    println!("{}", out);
    let mut out = format!("{:20}", "+␣time[ns]");
    for i in 0..stats.len() {
        out.push_str(&format!("{:^5.2}", stats[i].1));
        if i != stats.len() - 1 {
            out.push_str("|");
        }
    }
    println!("{}", out);
    let mut out = format!("{:20}", "-␣collisions");
    for i in 0..stats.len() {
        out.push_str(&format!("{:^5.2}", stats[i].2));
        if i != stats.len() - 1 {
            out.push_str("|");
        }
    }
    println!("{}", out);
    let mut out = format!("{:20}", "-␣time[ns]");
    for i in 0..stats.len() {
        out.push_str(&format!("{:^5.2}", stats[i].3));
        if i != stats.len() - 1 {
            out.push_str("|");
        }
    }
    println!("{}", out);
}

/// Writes data to csv file "hashset_data.csv"
///
/// Writes one file for all load factors combined.
/// Size of load_factors and all Vecs in all_stats must be the same
pub fn write_csv(all_stats: &[(String, Vec<(f32, f64, f32, f64)>)],
   ↪ load_factors: &[f64]) {
    let mut file = OpenOptions::new()
        .create(true)
        .write(true)
        .truncate(true)
        .open("hashset_data.csv")
        .expect("Could␣not␣open␣file␣to␣write␣output␣analysis␣to");
    let mut header = String::new();
    header.push_str("\"Name\",");
    for lambda in load_factors {
        let percentage = format!("{:.0}%", lambda * 100_f64);
        header.push_str(
            &format!("\"Success␣Collisions({0})\",\"Success␣Time({0})[ns
   ↪ ]\"\"Failures␣Collisions({0})\",\"Failures␣Time({0})[ns]\",",
   ↪ percentage));
    }
    header.push_str("\r\n");
    file.write_all(header.as_bytes())
        .expect("Could␣not␣write␣to␣file");
    for (name, stats) in all_stats {
        let mut f = format!("\"{}\"", name);
        for stat in stats {
            f.push_str(&format!(",{},{},{},{}", stat.0, stat.1, stat.2,
   ↪ stat.3));
        }
        f.push_str("\n");
        file.write_all(f.as_bytes())
            .expect("Could␣not␣write␣to␣file");
    }
}

/// Writes graph pngs in the graphs subfolder
///
/// Writes separate graphs for collision on success,
/// collisions on failure, time on success + time on failure
pub fn write_graphs(
    all_stats: &[(String, Vec<(f32, f64, f32, f64)>)],
    load_factors: &[f64],
    element_count: usize,
) {
    let mut fg = Figure::new();
    let ax = fg
        .axes2d()
        .set_title("Collisions␣on␣success", &[])
        .set_legend(Graph(0.5), Graph(0.9), &[], &[])
        .set_x_label("Number␣of␣elements", &[])
        .set_y_label("Collisions", &[]);
    for (name, stats) in all_stats {
        ax.lines(
            load_factors
                .iter()
                .map(|x| (x * element_count as f64) as usize),
            stats.iter().map(|x| x.0),
            &[Caption(&name)],
        );
    }
    fg.save_to_png("./graphs/successful_collisions.png", 1920, 1080)
        .expect("Could␣not␣save␣file");

    let mut fg = Figure::new();
    let ax = fg
        .axes2d()
        .set_title("Collisions␣on␣failure", &[])
        .set_legend(Graph(0.5), Graph(0.9), &[], &[])
        .set_x_label("Number␣of␣elements", &[])
        .set_y_label("Collisions", &[]);
    for (name, stats) in all_stats {
        ax.lines(
            load_factors
                .iter()
                .map(|x| (x * element_count as f64) as usize),
            stats.iter().map(|x| x.2),
            &[Caption(&name)],
        );
    }
    fg.save_to_png("./graphs/failure_collisions.png", 1920, 1080)
        .expect("Could␣not␣save␣file");

    let mut fg = Figure::new();
    let ax = fg
```

```
        .axes2d()
        .set_title("Time␣on␣success", &[])
        .set_legend(Graph(0.5), Graph(0.9), &[], &[])
        .set_x_label("Number␣of␣elements", &[])
        .set_y_label("time[ns]", &[]);
    for (name, stats) in all_stats {
        ax.lines(
            load_factors
                .iter()
                .map(|x| (x * element_count as f64) as usize),
            stats.iter().map(|x| x.1),
            &[Caption(&name)],
        );
    }
    fg.save_to_png("./graphs/successful_time.png", 1920, 1080)
        .expect("Could␣not␣save␣file");

    let mut fg = Figure::new();
    let ax = fg
        .axes2d()
        .set_title("Time␣on␣failure", &[])
        .set_legend(Graph(0.5), Graph(0.9), &[], &[])
        .set_x_label("Number␣of␣elements", &[])
        .set_y_label("time[ns]", &[]);
    for (name, stats) in all_stats {
        ax.lines(
            load_factors
                .iter()
                .map(|x| (x * element_count as f64) as usize),
            stats.iter().map(|x| x.3),
            &[Caption(name)],
        );
    }
    fg.save_to_png("./graphs/failure_time.png", 1920, 1080)
```

```
        .expect("Could␣not␣save␣file");
}
```