Aufgabe zu HashSets

Erik Imgrund, TINF19B1

20. Mai 2020

Inhaltsverzeichnis

Quellcodeverzeichnis

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.

2 Programm- bzw. Quellcode

2.1 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.

2.2 Vollständiger Quellcode

Listing 2.1: main.rs

```
extern crate gnuplot;
extern crate rand;

pub mod hashset;
use gnuplot::{AxesCommon, Caption, Figure, Graph};
use hashset::*;
use rand::{thread_rng, Rng};
use std::fs::OpenOptions;
use std::io::Write;
use std::time::Instant;
```

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

```
fn get_builder<T: PartialEq + 'static,␣H:␣'static + HashTable<T> + Default
) -> Box<dyn HashTableBuilder<T>> {
    Box::new(DefaultHashTableBuilder::<T, H>::new())
}

const RESIZE_TO_MAKE_FAIR: bool = true;
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,
    0.17, 0.18, 0.19, 0.2, 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28,
];
const ITERATIONS_PER_LOAD_FACTOR: usize = 50;

fn main() {
    #[rustfmt::skip]
    let tables: Vec<(Box<dyn HashTableBuilder<u32>>, String)> = vec![
        (get_builder::<u32, OpenAddressingTable::<u32, QuadraticProber, Mu
        (get_builder::<u32, OpenAddressingTable::<u32, QuadraticProber, Mo
        (get_builder::<u32, OpenAddressingTable::<u32, QuadraticProber, Xo
        (get_builder::<u32, OpenAddressingTable::<u32, LinearProber, MulHa
        (get_builder::<u32, OpenAddressingTable::<u32, LinearProber, ModHa
        (get_builder::<u32, OpenAddressingTable::<u32, LinearProber, XorSh
        (get_builder::<u32, OpenAddressingTable::<u32, TriangularProber, M
        (get_builder::<u32, OpenAddressingTable::<u32, TriangularProber, M
        (get_builder::<u32, OpenAddressingTable::<u32, TriangularProber, X
        (get_builder::<u32, DirectChainingTable::<u32, MulHash>>(), "Direc
        (get_builder::<u32, DirectChainingTable::<u32, ModHash>>(), "Direc
        (get_builder::<u32, DirectChainingTable::<u32, XorShiftHash>>(), "
        (get_builder::<u32, SeparateChainingTable::<u32, MulHash>>(), "Sep
        (get_builder::<u32, SeparateChainingTable::<u32, ModHash>>(), "Sep
        (get_builder::<u32, SeparateChainingTable::<u32, XorShiftHash>>(),
        (get_builder::<u32, CoalescedTable::<u32, MulHash>>(), "Coalesced␣
        (get_builder::<u32, CoalescedTable::<u32, ModHash>>(), "Coalesced␣
        (get_builder::<u32, CoalescedTable::<u32, XorShiftHash>>(), "Coale
    ];
    generate_stats(tables);
}

fn print_header(name: &str) {
    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);
}

fn print_subtable(name: &str, stats: &[(f32, f64, f32, f64)]) {
    println!();
    print_header(name);
    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);
}

fn generate_stats(tables: Vec<(Box<dyn HashTableBuilder<u32>>, String)>) {
    let mut all_stats = Vec::new();
    for (builder, name) in tables {
        let mut stats = [(0_f32, 0_f64, 0_f32, 0_f64); LOAD_FACTORS.len()]
        for (i, s) in LOAD_FACTORS.iter().enumerate() {
            for _ in 0..ITERATIONS_PER_LOAD_FACTOR {
                let temp = get_stats(builder.as_ref(), *s);
                stats[i].0 += temp.0;
                stats[i].1 += temp.1;
                stats[i].2 += temp.2;
                stats[i].3 += temp.3;
            }
            stats[i].0 /= ITERATIONS_PER_LOAD_FACTOR as f32;
            stats[i].1 /= ITERATIONS_PER_LOAD_FACTOR as f64;
            stats[i].2 /= ITERATIONS_PER_LOAD_FACTOR as f32;
            stats[i].3 /= ITERATIONS_PER_LOAD_FACTOR as f64;
        }
        print_subtable(&name, &stats);
        all_stats.push((name, stats));
    }

    // create output file for analysis in csv format
    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␣Ti
    }
    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, st
        }
        f.push_str("\n");
        file.write_all(f.as_bytes())
            .expect("Could␣not␣write␣to␣file");
    }

    // create graph for every type of HashTable
    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");
}

fn get_stats(builder: &dyn HashTableBuilder<u32>, fill: f64) -> (f32, f64,
    let fill = f64::min(fill * ELEMENT_COUNT as f64, ELEMENT_COUNT as f64)
    get_stats_rec(builder, fill, 0)
}

fn get_stats_rec(
    builder: &dyn HashTableBuilder<u32>,
    fill: usize,
    attempt: usize,
) -> (f32, f64, f32, f64) {
    let random_samples = 1_usize << 16;
    let repetitions_successful = 1;

    let mut table = builder.build();
    if RESIZE_TO_MAKE_FAIR {
        table.as_mut().resize_to_bytes(ELEMENT_COUNT << 3, fill);
    }
    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) {
            if attempt > 100 {
                return (std::f32::NAN, std::f64::NAN, std::f32::NAN, std::
            }
            return get_stats_rec(builder, fill, attempt + 1);
        }
    }
    let mut ns = 0_usize;
    let mut nf = 0_usize;
    let mut cs = 0_usize;
    let mut cf = 0_usize;
    let start_time = Instant::now();
    for x in &inserted_nums {
        table.as_mut().has(x);
    }
    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);
    }
    let duration_f = start_time.elapsed().as_nanos();
    for _ in 0..repetitions_successful {
        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());
            }
        }
    }
    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),
        (duration_s as f64 / fill as f64),
        (cf / nf),
        (duration_f as f64 / random_samples as f64),
    )
}
```

Listing 2.2: mod.rs

```
mod hashing;
mod probing;

pub use hashing::*;
pub use probing::*;
use std::collections::LinkedList;
use std::marker::PhantomData;

pub const ELEMENT_COUNT: usize = 1 << 15;

pub trait HashTable<T> {
    fn has(&mut self, val: &T) -> bool;
    fn reset_collisions(&mut self);
    fn get_collisions(&self) -> usize;
    fn insert(&mut self, val: &T) -> bool;
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize);
}

pub trait HashTableBuilder<T> {
    fn build(&self) -> Box<dyn HashTable<T>>;
}

pub struct DefaultHashTableBuilder<T: PartialEq, H: HashTable<T> + Default
    table: PhantomData<H>,
    t: PhantomData<T>,
}

impl<T: PartialEq, H: 'static␣+␣HashTable<T>␣+␣Default>␣HashTableBuilder<T
␣␣␣␣for␣DefaultHashTableBuilder<T,␣H>
{
␣␣␣␣fn␣build(&self)␣->␣Box<dyn␣HashTable<T>>␣{
␣␣␣␣␣␣␣␣Box::new(H::default())
␣␣␣␣}
}

impl<T:␣PartialEq,␣H:␣HashTable<T>␣+␣Default>␣DefaultHashTableBuilder<T,␣H
␣␣␣␣pub␣fn␣new()␣->␣Self␣{
␣␣␣␣␣␣␣␣Self␣{
␣␣␣␣␣␣␣␣␣␣␣␣table:␣PhantomData,
␣␣␣␣␣␣␣␣␣␣␣␣t:␣PhantomData,
␣␣␣␣␣␣␣␣}
␣␣␣␣}
}

pub␣struct␣DirectChainingTable<T:␣PartialEq␣+␣Copy,␣H:␣Hasher<T>>␣{
␣␣␣␣collisions:␣usize,
␣␣␣␣entries:␣Vec<LinkedList<T>>,
    hasher: PhantomData<H>,
}
impl<T: PartialEq + Copy, H: Hasher<T>> Default for DirectChainingTable<T,
    fn default() -> Self {
        Self::with_size(ELEMENT_COUNT)
    }
}

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

pub struct SeparateChainingTable<T: PartialEq + Copy, H: Hasher<T>> {
    collisions: usize,
    entries: Vec<(Option<T>, LinkedList<T>)>,
    hasher: PhantomData<H>,
}
impl<T: PartialEq + Copy, H: Hasher<T>> Default for SeparateChainingTable<
    fn default() -> Self {
        Self::with_size(ELEMENT_COUNT)
    }
}

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

// one Option<(T, Option<usize>)> has size 24 => Can only use 10.922 bucke
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> {
    fn default() -> Self {
        Self::with_size(ELEMENT_COUNT)
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> CoalescedTable<T, H> {
    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,
␣␣␣␣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
␣␣␣␣}
␣␣␣␣fn␣reset_collisions(&mut␣self)␣{
␣␣␣␣␣␣␣␣self.collisions␣=␣0;
```

```
␣␣␣␣}
␣␣␣␣fn␣get_collisions(&self)␣->␣usize␣{
␣␣␣␣␣␣␣␣self.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")
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self.entries[index]␣=␣Some((old,␣Some(self.cursor)));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣true;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣self.cursor␣+=␣1;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣true
```

```
␣␣␣␣}

␣␣␣␣fn␣resize_to_bytes(&mut␣self,␣bytes:␣usize,␣elements:␣usize)␣{
␣␣␣␣␣␣␣␣let␣entries␣=␣bytes␣/␣24;
␣␣␣␣␣␣␣␣if␣entries␣<␣elements␣{
␣␣␣␣␣␣␣␣␣␣␣␣panic!("cannot␣resize␣that␣low");
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣*self␣=␣Self::with_size(entries);
␣␣␣␣}
}

impl<T:␣PartialEq␣+␣Copy,␣H:␣Hasher<T>>␣HashTable<T>␣for␣SeparateChainingT
␣␣␣␣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;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣for␣x␣in␣&self.entries[index].1␣{
␣␣␣␣␣␣␣␣␣␣␣␣if␣*x␣==␣*val␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣true;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣self.collisions␣+=␣1;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣false
␣␣␣␣}
␣␣␣␣fn␣reset_collisions(&mut␣self)␣{
␣␣␣␣␣␣␣␣self.collisions␣=␣0;
␣␣␣␣}
␣␣␣␣fn␣get_collisions(&self)␣->␣usize␣{
␣␣␣␣␣␣␣␣self.collisions
␣␣␣␣}
```

```
    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_front(*val);
        true
    }
    // the type is kind of complicated for this hash table so some assumpt
    // a bucket has size 40, any element that collides with another one ta
    // if we assume that 0.25 of all elements have collided with another, 
    // 40*bytes + 0.25*24*elements
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize) {
        let available_bytes = bytes as isize - (elements as isize * 6);
        if available_bytes < 1 {
            panic!("invalid configuration for direct chaining table");
        }
        *self = Self::with_size(available_bytes as usize / 40);
    }
}

impl<T: PartialEq + Copy, H: Hasher<T>> HashTable<T> for DirectChainingTab
    fn has(&mut self, val: &T) -> bool {
        let index = H::hash(val, self.entries.len());
        for x in &self.entries[index] {
```

```
␣␣␣␣␣␣␣␣␣␣␣␣if␣*x␣==␣*val␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣true;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣self.collisions␣+=␣1;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣false
␣␣␣␣}
␣␣␣␣fn␣reset_collisions(&mut␣self)␣{
␣␣␣␣␣␣␣␣self.collisions␣=␣0;
␣␣␣␣}
␣␣␣␣fn␣get_collisions(&self)␣->␣usize␣{
␣␣␣␣␣␣␣␣self.collisions
␣␣␣␣}
␣␣␣␣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_front(*val);
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣true
␣␣␣␣}

␣␣␣␣//␣size␣of␣direct␣chaining␣table␣is␣buckets*(size␣of␣bucket)␣+␣entries
␣␣␣␣//␣size␣of␣bucket␣is␣the␣size␣of␣an␣empty␣linkedlist␣=␣24
␣␣␣␣//␣size␣of␣node␣is␣the␣size␣of␣a␣linkedlist␣node␣=␣24
␣␣␣␣fn␣resize_to_bytes(&mut␣self,␣bytes:␣usize,␣elements:␣usize)␣{
␣␣␣␣␣␣␣␣let␣available_bytes␣=␣bytes␣as␣isize␣-␣(24␣*␣elements)␣as␣isize;
␣␣␣␣␣␣␣␣if␣available_bytes␣<␣1␣{
␣␣␣␣␣␣␣␣␣␣␣␣panic!("not␣enough␣bytes␣available␣for␣the␣buckets");
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣*self␣=␣Self::with_size(available_bytes␣as␣usize␣/␣24);
␣␣␣␣}
}

//␣one␣Option<u32>␣has␣size␣of␣8B␣=>␣1␣<<␣15␣elements␣have␣(1␣<<␣18)B␣size
```

```
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 OpenAddress
    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 OpenA
    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
    }
    fn reset_collisions(&mut self) {
        self.collisions = 0;
    }
    fn get_collisions(&self) -> usize {
        self.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
    }
    fn resize_to_bytes(&mut self, bytes: usize, elements: usize) {
        if elements > ELEMENT_COUNT {
            panic!("trying to insert more elements than possible by constr
        }
        if bytes >> 3 != ELEMENT_COUNT {
            panic!("trying to resize to invalid size");
        }
    }
}
```

Listing 2.3: hashing.rs

```
pub struct ModHash;
impl Hasher<u32> for ModHash {
    fn hash(val: &u32, max: usize) -> usize {
        *val as usize % max
    }
}

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
    }
}

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 hashing trait
pub trait Hasher<T> {
    fn hash(val: &T, max: usize) -> usize;
}
```

Listing 2.4: probing.rs

```
pub trait Prober {
    fn probe(i: usize) -> usize;
}

pub struct TriangularProber;
impl Prober for TriangularProber {
    fn probe(i: usize) -> usize {
        (i * (i + 1)) >> 1
    }
}

pub struct LinearProber;
impl Prober for LinearProber {
    fn probe(i: usize) -> usize {
        i
    }
}

pub struct QuadraticProber;
impl Prober for QuadraticProber {
    fn probe(i: usize) -> usize {
        i * i
    }
}
```