Order-Statistic-Tree auf Basis eines Rot-Schwarz-Baums

Ziel der Datenstruktur

Gesucht ist ein dynamischer, geordneter Container, der:

- typische Operationen wie `insert`, `delete`, `search`, `min`, `max` unterstützt
- zusätzlich `predecessor` und `successor` anbietet
- zu jedem gefundenen Element seinen In-Order-Index (i) über alle Elemente inklusive Duplikaten zurückliefert

Warum ein Rot-Schwarz-Baum?

Für die Basis kommt nur eine balancierte Suchbaumstruktur in Frage, um alle Operationen in $O(\log n)$ zu garantieren. Ein Rot-Schwarz-Baum bietet hier einen guten Kompromiss:

- Die Höhe bleibt immer $O(\log n)$, sodass Einfügen, Löschen und Suchen logarithmisch bleiben.
- Die Implementierung der Rot-Schwarz-Eigenschaften (Farben, Rotationen, Fixups) ist gut dokumentiert und es gibt reichlich Beispiele zur Orientierung.

Einfache binäre Suchbäume können im Durchschnitt zwar schnell sein, werden aber bei ungünstiger Einfügefolge zu einer Liste und verlieren dann die logarithmischen Garantien. AVL-Bäume wären eine Alternative, sind aber "hungriger". Da Rot-Schwarz-Bäume hier bereits ausreichend balanciert sind und mir selber auch vertrauter waren, fiel die Wahl leicht.

Warum „Order-Statistic-Tree“?

Ein Order-Statistic-Tree (OST) ist ein balancierter Suchbaum, der zusätzlich in jedem Knoten die Groeße seines Teilbaums speichert.

Damit lassen sich Rang-Operationen wie

- „Wie viele Elemente sind kleiner als `x`?“
- „Welches Element steht an Rang `i` in der sortierten Folge?“

in $O(\log n)$ beantworten.

Die vorliegende Implementierung erweitert den Rot-Schwarz-Baum genau in diesem Sinne:

- Jeder Knoten kennt die Größe seines Teilbaums (`size`) und die Anzahl gleicher Schlüssel (`count`).
- `min`, `max`, `predecessor`, `successor` geben neben dem Schlüssel auch seinen In-Order-Index zurück.

Operationen der Datenstruktur

- `insert(key, value)`: Fuegt ein weiteres Vorkommen von `key` mit zugehoerigem `value` ein; existiert `key` bereits, wird zunaechst nur `count` in diesem Knoten erhoeht, sonst wird ein neuer Knoten mit `count = 1` in den Baum eingefuegt.
- `delete(key)`: Entfernt ein Vorkommen von `key`, falls vorhanden; bei `count > 1` wird lediglich `count` dekrementiert, bei `count = 1` wird der entsprechende Knoten wie in einem Rot-Schwarz-Baum geloescht und der Baum neu balanciert.
- `count(key) → ℕ`: Liefert die Anzahl des Schluessels `key` (0, falls `key` nicht enthalten ist).
- `size() → ℕ`: Liefert die Gesamtzahl aller Elemente im Baum, also die Summe aller `count`-Werte ueber alle Knoten.
- `search(key) → bool`: Prueft, ob `key` mindestens einmal im Baum vorkommt (`count(key) > 0`).
- `isEmpty() → bool`: Prueft, ob `size() = 0` gilt.
- `min()/max() → <i, key>`: Liefert kleinstes/groesstes vorkommendes `key` samt In-Order-Index `i` ueber alle Elemente inklusive Duplikaten.
- `predecessor(z) → <i, key>`: Liefert das groesste Element mit `key <= z` sowie den Rang `i` des letzten Vorkommens dieses Schluessels in der In-Order-Reihenfolge.
- `successor(z) → <i, key>`: Liefert das kleinste Element mit `key >= z` sowie den Rang `i` des ersten Vorkommens dieses Schluessels in der In-Order-Reihenfolge.

Aufbau des Baums und der Knoten

Knotentyp

Der Knotentyp ist in `node.zig` generisch über den Schlüsseltyp `T` definiert:

```
pub fn Node(comptime T: type) type {
    return struct {
        const Self = @This();
        data: T,

        // Kinder und Elternzeiger (klassischer Rot-Schwarz-Baum)
        left: ?*Self,
        right: ?*Self,
        parent: ?*Self,

        // Order-Statistic-Erweiterung
        size: usize, // Anzahl Elemente im Teilbaum (inkl. Duplikate)
        count: usize, // Multiplizität dieses Schlüssels in diesem Knoten

        // Rot-Schwarz-Farbe
        color: Color,
    };
}
```

- `data` ist der gespeicherte Schlüssel (es gibt keinen separaten Value-Typ – der Baum verwaltet allein die Schlüssel).
- `left`, `right`, `parent` bilden die Baumstruktur.
- `size` zählt alle Elemente im Teilbaum: `size = size(left) + size(right) + count` .
- `count` ist die Anzahl gleicher Schlüssel in genau diesem Knoten (Multimengen-Semantik).
- `color` ist `Red` oder `Black` und implementiert die Rot-Schwarz-Invarianten.

Baumtyp und Konfiguration

Der eigentliche Baumtyp wird in `root.zig` über eine Zig GenericFactory erzeugt:

```
pub const OrderStatisticTreeConfig = struct {
    use_freelist: bool = true,
    compact_sizes: bool = false,
};
```

```
pub fn OrderStatisticTree(
    comptime T: type,
    comptime compareFn: fn (a: T, b: T) std.math.Order,
    comptime cfg: OrderStatisticTreeConfig,
) type { ... }
```

- `T` ist der Schlüsseltyp.
- `compareFn` definiert die Ordnungsrelation (`lt`, `eq`, `gt`).
- `cfg` steuert zwei Optimierungen:
  - `use_freelist` : aktiviert einen Freelist-Speicherpool für Knoten.
  - `compact_sizes` : speichert Teilbaumgroeßen intern in einem kompakten Typ (`u32`), um Knoten kleiner zu machen und den Cache besser auszunutzen

Unterstützte Operationen und Laufzeiten

Die API des Baums bietet aktuell:

- `init(allocator) / deinit()` : Erzeugt bzw. zerstört einen Baum.
- `insert(data: T) !void` : Fügt ein Element ein.
  - Falls `data` bereits existiert, wird nur `count` des Knotens erhöht.
  - Andernfalls wird ein neuer Rot-Schwarz-Knoten angelegt und der Baum per `insertFixup` rebalanziert.
- `delete(data: T) void` : Entfernt ein Vorkommen von `data` .
  - Bei `count > 1` wird nur `count` dekrementiert und entlang des Pfades `size` angepasst.
  - Bei `count == 1` wird der physische Knoten wie in einem Rot-Schwarz-Baum entfernt und mit `deleteFixup` balanciert.
- `search(data: T) bool` : Prüft, ob `data` mindestens einmal enthalten ist.
- `isEmpty() bool` : Prüft, ob der Baum leer ist.
- `min() ?NodeResult` : Liefert das kleinste Element und dessen In-Order-Index (immer 0).
- `max() ?NodeResult` : Liefert das größte Element und den Index des **ersten** Vorkommens dieses Schlüssels in der In-Order-Reihenfolge.
- `predecessor(data: T) ?NodeResult` :
  - Sucht das größte Element, das **strikt kleiner** als `data` ist (also `< data`).
  - Gibt Schlüssel und In-Order-Index des ersten Vorkommens dieses Schlüssels zurück.
- `successor(data: T) ?NodeResult` :
  - Sucht das kleinste Element, das **strikt größer** als `data` ist (also `> data`).
  - Gibt Schlüssel und In-Order-Index des ersten Vorkommens dieses Schlüssels zurück.

Die In-Order-Indizes werden aus den gespeicherten `size`-Werten berechnet:

- `sizeOf(node)` liefert die Größe eines Teilbaums (0 bei `null`).
- `getRank(node)` läuft von einem Knoten bis zur Wurzel und addiert:
  - die Größen der linken Teilbäume dort, wo der aktuelle Knoten im rechten Kind hängt
  - plus die jeweiligen `count`-Werte.

Laufzeit und Speicherbedarf

Durch die Rot-Schwarz-Eigenschaften bleibt die Baumhöhe in O(log n), womit gilt:

- `insert`, `delete`, `search`, `min`, `max`, `predecessor`, `successor` laufen in O(log n).
- Die Zusatzarbeit für `size`-Aktualisierungen ist auf den Pfad von der Einfüge-/Löschstelle zur Wurzel beschränkt, also ebenfalls O(log n).
- Der Speicherbedarf ist O(log n) in der Anzahl der gespeicherten Elemente (inkl. Duplikate). Jeder unterscheidbare Schlüssel benötigt einen Knoten mit
  - drei Zeigern (`left`, `right`, `parent`),
  - einem Schlüssel `data`,
  - Farbe `color`,
  - Metadaten `size` und `count`.

Die Konfiguration `compact_sizes = true` verkleinert den Typ, in dem `size` intern geführt wird (z.B. auf `u32`). Das reduziert die Knotengröße und verbessert Lokalität, setzt aber voraus, dass die maximale Teilbaumgröße nicht über diesen Typ hinauswächst.

Implementierungsdetails und Optimierungen

Balancierung und Fixups

Die Rot-Schwarz-Invarianten werden mit klassischen Fixup-Prozeduren sichergestellt:

- `insertFixup` behandelt die bekannten Fälle (Onkel rot/schwarz, Doppelrot) und führt Rotationen (`rotateLeft` / `rotateRight`) sowie Umfärbungen durch, bis die Invarianten wieder gelten.
- `deleteFixup` sorgt nach einer Löschung dafür, dass die Schwarzhöhe konsistent bleibt (Behandlung von „Double-Black"-Fällen mit passender Fallunterscheidung und Rotationen).

Die Rotationen rufen immer `updateSize` auf den beteiligten Knoten auf, um die `size`-Information nach der strukturellen Änderung lokal zu reparieren.

Freelist zur Beschleunigung von Insert/Delete

Optional kann ein Freelist-Speicherpool pro Baum aktiviert werden:

- Beim Löschen wandern Knoten nicht direkt zum Allocator zurück, sondern in eine einfache Singly-Linked-Liste (per `right`-Zeiger).
- Beim Einfügen wird zuerst versucht, einen Knoten aus dieser Freelist zu recyceln; erst wenn sie leer ist, wird ein neuer Knoten beim Allocator angefordert.
- Das reduziert Allokationen/Deallokationen und verbessert die Laufzeit insbesondere in Szenarien mit vielen Lösch-/Insert-Zyklen („churn").

In Benchmarks zeigte der Freelist-Modus eine deutliche Verbesserung im zyklischen Delete/Insert-Szenario gegenüber der reinen Allocator-Variante.

Kompakte Größenrepräsentation

Mit `compact_sizes = true` wird der interne Typ für Teilbaumgrößen auf einen kleineren Integer-Typ abgebildet (z.B. `u32`), und an allen Stellen, wo Indizes nach außen als `usize` benötigt werden, findet eine explizite Konvertierung statt.

- Vorteil: kleinere Knoten, bessere Cache-Nutzung.
- Voraussetzung: maximale Teilbaumgröße bleibt im Bereich dieses Typs.

Micro-Optimierungen im Hot-Path

Einige kleine Optimierungen zielen direkt auf den „Hot-Path" der Baumoperationen:

- In `insert` wird das Ergebnis von `compareFn(data, node.data)` zwischengespeichert und nicht mehrfach pro Ebene berechnet.
- `sizeOf` und `updateSize` sind `inline`, damit der Compiler sie in die relevanten Pfade integrieren kann.
- Rotationen aktualisieren nur die lokal betroffenen Knoten (`updateSize`), während `updatePathSize` gezielt dort eingesetzt wird, wo der gesamte Pfad nach oben neu gerechnet werden muss.

Tests und Validierung

`tests.zig` enthält eine Reihe von Tests, die die Korrektheit der Implementierung absichern:

- Basisfälle (leerer Baum, einfache Inserts/Searches).
- `min`, `max` und In-Order-Traversierung über `successor`, inklusive Prüfung der Indizes.
- `predecessor` und `successor` für vorhandene und nicht vorhandene Schlüssel.
- Verhalten bei Duplikaten (richtige Indizes, korrekte Behandlung von `count` beim Löschen).
- Unterschiedliche Konfigurationen (`compact_sizes = false/true`) werden mit demselben Testsatz geprüft.

Quellen / Hilfsquellen

- https://www.geeksforgeeks.org/dsa/time-complexities-of-different-data-structures/
- https://en.wikipedia.org/wiki/Predecessor_problem
- https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

- https://en.wikipedia.org/wiki/Van_Emde_Boas_tree
- https://ls2-web.cs.tu-dortmund.de/~mamicoja/dap2/slides/lec_redblack.pdf
- https://groups.csail.mit.edu/mac/projects/info/schemedocs/ref-manual/html/scheme_113.html
- https://dtu.ac.in/Web/Departments/CSE/faculty/lect/DSA_MK_Lect8.pdf
- https://www.baeldung.com/cs/skip-lists
- https://lib.rs/crates/indexset
- https://docs.rs/order-stat/latest/order_stat/
- https://github.com/CogitatorTech/ordered?tab=readme-ov-file
- https://www.cs.yale.edu/homes/aspnes/pinewiki/OrderStatisticsTree.html
- https://en.wikipedia.org/wiki/Order_statistic_tree

Zum "ausprobieren":

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html