Datenstrukturen und Algorithmen
Übung 6 – Hashing

Abgabefrist: 07.04.2022, 16:00 h

Verspätete Abgaben werden nicht bewertet.

Theoretische Aufgaben

1. Betrachten Sie eine Hashtabelle der Grösse $m = 512$ und eine zugehörige Hashfunktion

   $$h(k) := \lfloor m(kA \bmod 1) \rfloor,$$

   wobei

   $$A := (\sqrt{5} - 1)/2.$$

   (a) Berechnen Sie die Plätze, auf die die Schlüssel 2022, 2023, 2024, 2025 abgebildet werden.

   (b) Finden Sie einen Schlüssel, der eine Kollision mit dem Schlüssel 2021 auslöst und gleichzeitig eine Postleitzahl von Bern ist.

   (1 Punkt)

2. Professor Joe behauptet, dass eine erhebliche Performanzsteigerung erzielt werden kann, wenn wir das Verkettungsschema so modifizieren, dass jede Liste in sortierter Ordnung gehalten wird. Wie beeinflusst die durch den Professor vorgeschlagene Änderung die Laufzeit für erfolgreiches Suchen, erfolgloses Suchen, Einfügen und Löschen? **(1 Punkt)**

3. Betrachten Sie das Einfügen der Schlüssel 24, 18, 13, 56, 44, 7, 19, 23, 33 in eine Hashtabelle der Länge $m = 11$ durch offene Adressierung mit der Hilfshashfunktion $h'(k) = k \bmod m$. Illustrieren Sie das Ergebnis des Einfügens dieser Schlüssel mithilfe von:

   (a) linearem Sondieren

   (b) quadratischem Sondieren mit $c_1 = 1$ und $c_2 = 3$

   (c) doppeltem Hashing mit $h_2(k) = 1 + (k \bmod (m - 1))$

   (1 Punkt)

4. Bei der Kollisionsauflösung nach dem Verkettungsschema muss für kollidierende Elemente neuer Speicher alloziert werden, bevor sie an den Kopf der Liste an ihrem Tabellenplatz gesetzt werden können. Machen Sie einen Vorschlag, wie dazu Speicherplatz innerhalb der Tabelle genutzt, also zugewiesen und freigegeben werden kann. Alle unbenutzten Tabellenplätze sollen in einer doppelt verlinkten Liste geführt werden (einer Freiliste). Diese Freiliste soll direkt innerhalb der Tabelle implementiert sein. Nehmen Sie dazu an, dass auf jedem Tabellenplatz eine Markierung (frei/benutzt) und entweder **a)** ein Element und ein Zeiger oder **b)** zwei Zeiger abgespeichert werden können:

   $$\langle flag, element, ptr \rangle \text{ oder } \langle flag, ptr_1, ptr_2 \rangle.$$

   Alle Operationen (Einfügen, Löschen, erfolgreiches sowie erfolgloses Suchen) sollten in einer erwarteten Zeit von $\mathcal{O}(1 + \alpha)$ ablaufen (keine Analyse notwendig).

   Muss die Freiliste dafür doppelt verkettet sein oder genügt eine einfach verkettete Liste? **(2 Punkte)**

Praktische Aufgaben

In dieser Übung werden Sie eine Variante von Hashing entwickeln, um räumliche Daten zu verwalten. Als Ausganslage dazu stellen wir auf *Ilias* Code zur Verfügung, der eine einfache Partikelsimulation implementiert. Nehmen Sie sich ein paar Minuten Zeit, um den Code zu studieren. Der Code simuliert eine Menge von Partikeln, die sich frei im zwei-dimensionalen Raum bewegen, bis sie mit einem anderen Partikel oder mit der Umgebung kollidieren. Bei einer Kollision werden die Partikel gemäss einem einfachen Modell abgelenkt. Die Simulation erfolgt über diskrete Zeitschritte. In jedem Schritt werden die Partikel gemäss ihrer Geschwindigkeit weiterbewegt, und bei Kollisionen wird zusätzlich die neue Geschwindigkeit und Richtung berechnet.

Eine naive Implementation der Kollisionsdetektion überprüft jedes Paar von Partikeln auf eine mögliche Kollision. Der Aufwand für die Kollisionsdetektion ist somit $\mathcal{O}\left(n^2\right)$, wobei n die Anzahl Partikel ist. Bei einer grösseren Anzahl Partikel wird dadurch der Aufwand zur Berechnung jedes Schritts in der Simulation schnell von der Kollisionsdetektion dominiert.

Ihre Aufgabe ist es, einen effizienteren Algorithmus zur Kollisionsdetektion zu entwickeln. Die Idee ist es, eine Datenstruktur zu verwenden, welche die Partikel gemäss ihrer räumlichen Position verwaltet. Weil die Partikel zufällig im Raum verteilt sind, kann die Position eines Partikels verwendet werden, um einen Hashwert zu berechnen ähnlich wie zu Beginn von Kapitel 11.3 im Buch beschrieben. Gegeben die x-Koordinate des Partikels im Bereich $0 \leq x < w$ ist der Hashwert $h(x) = \lfloor xm/w \rfloor$, wobei m die Grösse der Hashtabelle ist. Ebenso kann für die y-Koordinate ein Hashwert berechnet werden. Die beiden Hashwerte können nun als Indizes in eine zweidimensionale Hashtabelle verwendet werden.

Die zweidimensionale Hashtabelle entspricht einem zweidimensionalen Gitter: jeder Partikel wird entsprechend seiner Position einer Gitterzelle zugeordnet. Damit kann die Kollisionsdetektion effizienter gemacht werden, indem jeder Partikel nur auf Kollisionen in seiner eigenen Gitterzelle und den unmittelbaren Nachbarzellen untersucht wird. Die Nachbarzellen müssen getestet werden, weil Partikel eine gewisse Ausdehnung haben und mit den Nachbarzellen überlappen können. Beachten Sie auch, dass die Partikel nach jedem Simulationsschritt entsprechend ihrer neuen Position aus der Hashtabelle entfernt und in die richtige Zelle wieder eingetragen werden müssen.

1. Modifizieren Sie die Klasse `BouncingBallsSimulation`, um den skizzierten Algorithmus zu implementieren. Verwenden Sie eine zweidimensionale ArrayList von verketteten Listen als Hashtabelle. Für die Listen können Sie die Java `LinkedList` Klasse verwenden. Ihre Hashtabelle ist dann vom Typ `ArrayList<ArrayList<LinkedList<Ball>>>` [1] . Verwenden Sie einen Iterator vom Typ `ListIterator<Ball>` um die Listen zu traversieren. Dieser Iterator erlaubt Ihnen mit seiner Methode `remove()` effizient das zuletzt besuchte Element aus der Liste zu entfernen.

   Drucken Sie nur Ihre modifizierte Klasse `BouncingBallsSimulation` aus und geben Sie sie ab (Falls sie anderen Code verändert haben, dann geben sie den bitte auch ab — das sollte aber nicht nötig sein). **(3 Punkte)**

2. Testen Sie Ihren Algorithmus für verschiedene Grössen der Hashtabelle und verschiedene Anzahl Partikel. Stellen Sie die Zeitmessungen in einer Tabelle zusammen. Was ist jeweils die optimale Grösse der Hashtabelle? Was könnte der Grund sein, dass die Geschwindigkeit bei grösseren Tabellen wieder abnimmt? **(2 Punkte)**

Vergessen Sie nicht Ihren Sourcecode innerhalb der Deadline über die Ilias Aufgabenseite einzureichen.

[1] Sie dürfen auch stattdessen eine einzige lineare `ArrayList<LinkedList<Ball>>` verwenden, bspw. mit dem Index $h_y \cdot m + h_x$