Teil VII

Hashverfahren

Überblick

Hashverfahren: Prinzip

- Ziel: Dictionary mit Kosten $O(1)$
- Speicherung in Feld 0 bis $m-1$, einzelne Positionen oft als „Buckets“ bezeichnet
- Hashfunktion $h(e)$ bestimmt für Element e die Position im Feld
- h sorgt für „gute“, kollisionsfreie bzw. kollisionsarme Verteilung der Elemente
- letzteres kann nur annäherungsweise durch Gleichverteilung erreicht werden, da Elemente vorher unbekannt!

Beispiel für Hashen

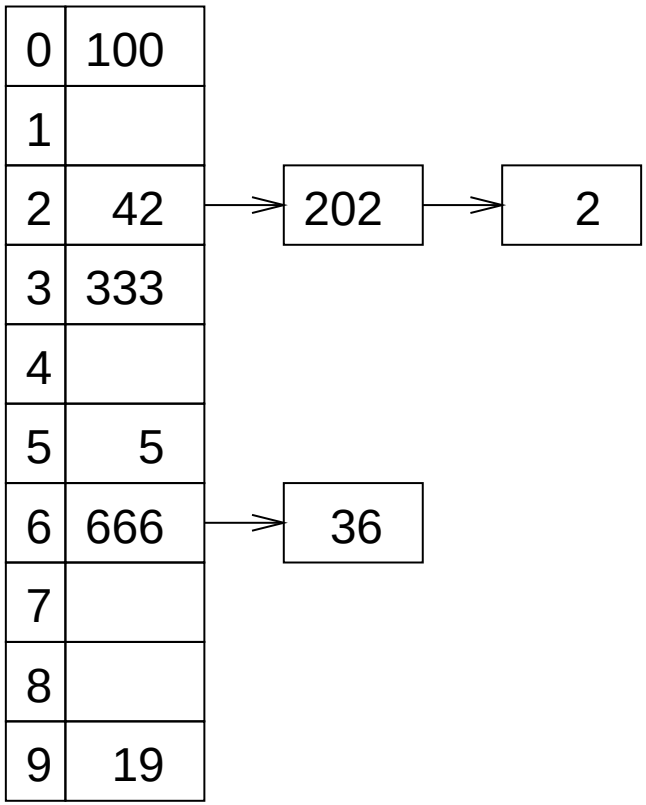
- Beispiel: Array von 0 bis 9, $h(i) = i \mod 10$
- Array nach Einfügen von 42 und 119:

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

- Möglichkeit der *Kollision*: Versuch des Eintragens von 69

Kollisionsstrategie: Verkettung der Überläufer

- Idee: alle Einträge einer Array-Position werden in einer verketteten Liste verwaltet



Grundbegriffe Hashing

- U = Universum = Menge aller zulässigen Schlüssel
- n = Anzahl der aktuell gespeicherten Datensätze, meist $n \ll |U|$
- m = Größe der Hashtabelle, häufig $m \approx n$
- $h\colon U \to \{0, \ldots, m-1\}$
- $h(key)$ heißt Hashwert

- in Hashtabelle werden Datensätze oder Verweise darauf gespeichert
- Annahme: keine Duplikate

Grundlagen: Hashfunktionen

- hängen vom Datentyp der Elemente und konkreter Anwendung ab
- für Integer oft

$$h(i) = i \mod m$$

(funktioniert gut, wenn m eine Primzahl ist)

- für andere Datentypen: Rückführung auf Integer
  - Fließpunkt-Zahlen: Addiere Mantisse und Exponent
  - Strings: Addiere ASCII/Unicode-Werte der/einiger Buchstaben, evtl. jeweils mit Faktor gewichtet

$$h(s) = (a_0 \cdot s[0] + a_1 \cdot s[1] + \cdots + a_{l-1} \cdot s[l-1]) \mod m$$

  - Sonstige: Bitdarstellungals Zahl interpretieren. (Es sollte $m \neq 2^i$ gelten, sonst Abhängigkeit von nur wenigen Bits.)

Hashfunktionen: Anforderungen

- sollen die Werte gut „streuen“: gleichmäßige Verteilung der real vorkommenden Schlüssel auf $\{0, \ldots, m-1\}$
- eventuell von Besonderheiten der Eingabewerte abhängig (Buchstaben des Alphabets in Namen tauchen unterschiedlich häufig auf!)
- sind effizient berechenbar (konstanter Zeitbedarf, *auf keinen Fall* abhängig von der Anzahl der gespeicherten Werte!)

Ungünstige Hashfunktionen

- Beispiel #1:
  - $N = 2^i$ und generierte Artikelnummern mit Kontrollziffer 1, 3 oder 7 am Ende
    $\rightsquigarrow$ Abbildung nur auf ungerade Adressen

- Beispiel #2:
  - Matrikelnummern in Hashtabelle mit 100 Einträgen
  - Variante a: erste beiden Stellen als Hashwert
    $\rightsquigarrow$ Abbildung auf wenige Buckets
  - Variante b (besser): letzte beiden Stellen
    $\rightsquigarrow$ gleichmäßige Verteilung

Kollisionsstrategien

Hashing mit Verkettung (engl.: chaining)

- Variante 1: erster Eintrag direkt in Hashtabelle, weitere in Liste
- Variante 2: Hashtabelle enthält nur Verweise auf Listen
  - Spezialfallbehandlung entfällt
  - in Hashtabelle muss kein Speicherplatz für Datensätze reserviert werden
- statt Listen sind auch andere Datenstrukturen (z.B. Baum) möglich

Offene Adressierung/Sondierungsverfahren

- bei Kollision anderen Platz innerhalb der Hashtabelle suchen
- z.B. $h(k) \rightarrow h(k)+1 \rightarrow h(k)+2 \rightarrow \ldots$
- Problem: Delete erzeugt Lücke. Bei Search dort abbrechen?

Bsp.: Lineares Sondieren

	leer	insert 89	insert 18	insert 49	insert 58	insert 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Aufwand beim Hashen

- bei geringer Kollisionswahrscheinlichkeit:
  - Suchen in $O(1)$
  - Einfügen in $O(1)$
  - Löschen bei Sondierverfahren: nur Markieren der Einträge als gelöscht $O(1)$, oder „Rehashen“ der gesamten Tabelle notwendig $O(n)$
- Füllgrad über 80 % : Einfüge- / Suchverhalten wird schnell dramatisch schlechter aufgrund von Kollisionen (bei Sondieren)

Aufwand beim Hashen: Verkettete Überläufer

- Füllgrad/Belegungsgrad (load factor) $\alpha = n/m$
  - n Anzahl gespeicherter Elemente
  - m Anzahl Buckets
  - = durchschnittliche Länge der Überlauflisten

- erfolglose Suche (Hashing mit Überlaufliste): $O(1 + \alpha)$
- erfolgreiche Suche ebenfalls in dieser Größenordnung (bei Annahme der Gleichverteilung)
- worst case: $O(n)$

Anwendung in Java Collection Framework

- `Object` enthält `int hashCode()`
  - liefert Adresse des Objekts
  - kann in Unterklassen überschrieben werden
  - Abstimmung mit Methode equals notwendig (gleich $\Rightarrow$ gleicher Hashwert)
- Interface `Set` wird implementiert durch
  - `HashSet<E>` (Implementierung durch Hashing) und
  - `TreeSet<E>` (Implementierung durch ausgeglichenen Suchbaum)
- Interface `Map<K, V>` (Schlüssel/Wert-Paare) wird implementiert durch
  - `HashMap<K, V>` (Hashing wird auf Schlüssel angewandt)
  - `TreeMap<K, V>` (Implementierung durch ausgeglichenen Suchbaum)

Zusammenfassung

- Berechnung der „Adresse“ eines Eintrags in einer Datenstruktur statt Suchen
- erfordert geeignete Hashfunktion
- Menge möglicher Elemente $>$ Anzahl verfügbarer Positionen
  $\rightsquigarrow$ Kollisionsbehandlung
- Problem: feste Größe des Bildbereichs
  $\rightsquigarrow$ dynamische Hashverfahren
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukturen*, Kap. 15

Zusammenfassung: ADT und Datenstrukturen