Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Hashtabellen und Überläufer

Umgang mit Kollision und Überläufer

Folgende Verfahren können eingesetzt werden wenn eine Kollision festgestellt wird, d.h. der mit $h(k_1)$ berechnete Platz ist bereits mit einem Schlüssel k_2 mit $k_1 \neq k_2$ belegt, d.h. das einzufügende Element wird zum Überläufer

1. Verkettung der Überläufer mit Listen - man bringt die Überläufer außerhalb der Tabelle unter
2. Offene Hashverfahren - man bringt die Überläufer in der Tabelle woanders unter:
   - Lineares Sondieren
   - Quadratisches Sondieren
   - Double Hashing
   - Ordered Hashing
   - Robin-Hood Hashing
   - Coealesced Hashing
3. Dynamische Hashverfahren - Hashverfahren für stark wachsende oder schrumpfende Datenbestände

Umgang mit Kollision und Überläufer

Werden zwei Datensätze dem gleichen Bucket zugeteilt, dann spricht man von einer Kollision. Wir benötigen also Methoden um Kollisionen zu behandeln. Die führt zu folgender Idee:

Zusammegefasst gibt es also zwei Lösungsansätze:

1. Jede Tabellenzelle wird um eine Liste oder Suchbaum erweitert, in denen die kollidierenden Datensätze untergebracht werden.

2. Oder: Im Fall einer Kollision such man nach einer alternativen Position. Diese Technik nennt man Sondieren.

Verkettung der Überläufer

Sei N = 10 und h(x) = x mod 10. Werden die Datensätze 100, 42, 5, 333, 19, 666, 202, 36 und 2 in eine Hashtabelle eingetragen, so ergibt sich folgende Situation:

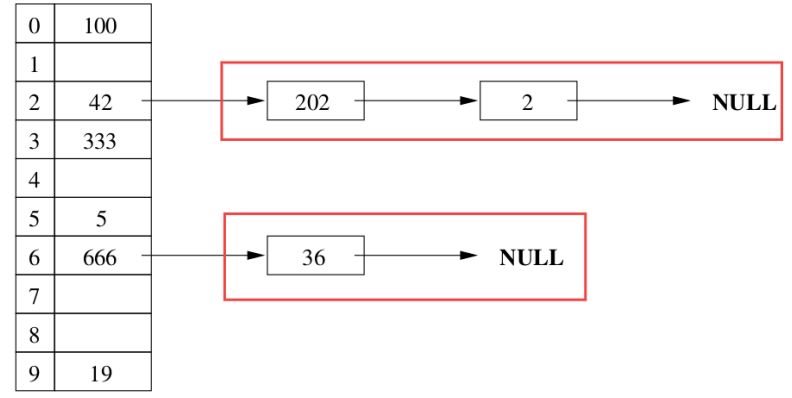

Abb. 1. Beispiel für die Kollisionsbehandlung mit linearen Listen

Verkettung der Überläufer

Zwei Methoden können angewendet werden:

1. **Separate Verkettung der Überläufer**
2. **Direkte Verkettung der Überläufer**

Gegeben sei eine Hashtabelle *t* der Größe *N* mit einer Hashfunktion h(k): k -> index ∈ [0,N-1].

Separate Verkettung der Überläufer

- Jedes Element der Hashtabelle t ist Anfangselement einer Überlaufkette (verkettete lineare Liste).

[Widmayer]

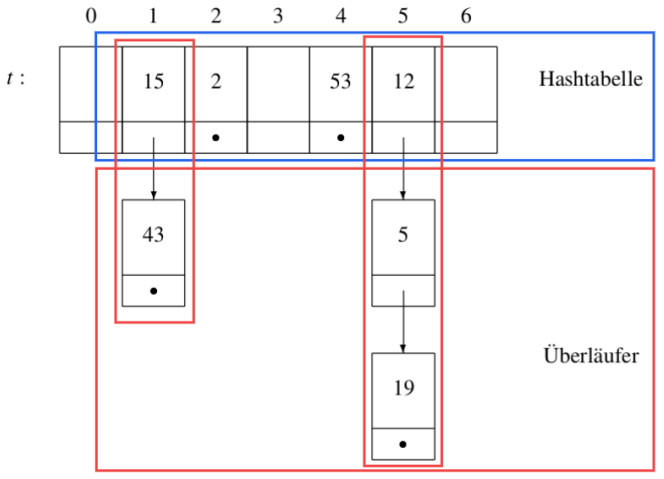

Abb. 2. Mix aus Hashtabelleneintrag und Verkettung

Separate Verkettung der Überläufer

- **Suchen** nach Schlüssel k: Beginne bei t[h(k)] und folge den Verweisen der Überlaufkette, bis entweder k gefunden wurde (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist (erfolglose Suche).
- **Einfügen** eines Schlüssels k: Suche nach k; die Suche verläuft erfolglos (sonst wird k nicht eingefügt) und endet am Ende einer Überlaufkette oder bei t[h(k)].
  - Im letzteren Fall trage k in t[h(k)] ein; sonst erzeuge ein neues Listenelement und hänge es ans Ende der Überlaufkette an.
- **Entfernen** eines Schlüssels k: Suche nach k; die Suche verläuft erfolgreich (sonst kann k nicht entfernt werden).
  - Steht k in der Hashtabelle, so streiche k dort; falls eine Überlaufkette bei t[h(k)] beginnt, so übertrage das erste Element der Überlaufkette nach t[h(k)] und entferne es aus der Überlaufkette.
  - Steht k in einem Element der Überlaufkette, so entferne dieses Element aus der Überlaufkette.

Separate Verkettung der Überläufer

Allerdings fällt auf, daß die unterschiedlichen Fälle (Schlüssel in Hashtabelle oder in Überlaufkette, gegebenenfalls Nachziehen des ersten Überläufers in die Hashtabelle beim Entfernen, usw.) einige Abfragen erfordern, die die Laufzeit der Operationen spürbar beeinträchtigen.

Separate Verkettung der Überläufer

Allerdings fällt auf, daß die unterschiedlichen Fälle (Schlüssel in Hashtabelle oder in Überlaufkette, gegebenenfalls Nachziehen des ersten Überläufers in die Hashtabelle beim Entfernen, usw.) einige Abfragen erfordern, die die Laufzeit der Operationen spürbar beeinträchtigen.

Wenn man bereit ist, unter Umständen etwas Speicherplatz zu opfern, so kann man auch einfach alle Datensätze in den Überlaufketten speichern; in der Hashtabelle benötigt man dann nur Zeiger auf den Listenanfang.

Direkte Verkettung der Überläufer

Jedes Element der Hashtaelle ist ein Zeiger auf eine (Überlauf-) Kette. Im wesentlichen handelt es sich hierbei also stets um Operationen in linearen ver-ketteten Listen.

[Widmayer]

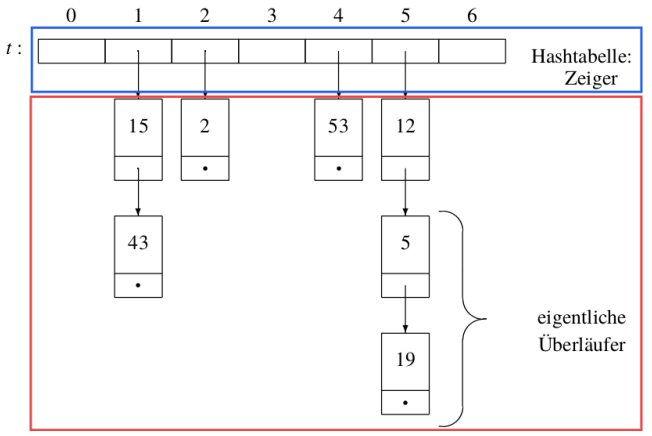

Abb. 3. Vereinfachtes Verfahren: Alle Elemente werden immer in Listen gespeichert. Der Hasheintrag ist nur Referenz auf Liste.

Direkte Verkettung der Überläufer

- **Suchen** nach Schlüssel k: Beginne bei el=↑t[h(k)] und folge den Verweisen der Überlaufkette, bis entweder k gefunden wurde (erfolgreiche Suche) oder das Ende der Überlaufkette erreicht ist (erfolglose Suche).
- **Einfügen** eines Schlüssels k: Suche nach k; die Suche endet erfolglos am Ende einer Überlaufkette (sonst wird k nicht eingefügt). Schaffe ein neues Listenelement und hänge es ans Ende der Überlaufkette an.
- **Entfernen** eines Schlüssels k: Suche nach k; die Suche verläuft erfolgreich (sonst kann k nicht entfernt werden) und endet bei einem Element der Überlaufkette. Entferne dieses Element aus der Überlaufkette.

Direkte Verkettung der Überläufer

Die durchschnittliche Anzahl der Einträge in einer Kette ist gerade $\alpha = n / m$, wenn n Einträge auf m Ketten verteilt sind.

Die Effizienz der erfolglosen Suche läßt sich verbessern, wenn man die Überlaufketten sortiert hält. Dann muß man beim erfolglosen Suchen nicht stets bis zum Listenende suchen, sondern kann im Mittel schon in der Mitte der Liste der Überläufer aufhören (man beachte, daß die erfolglose Suche dann für $\alpha \leq 1$ im Mittel schneller ist als die erfolgreiche).

- Nach den angegebenen Methoden der direkten und separaten Verkettung der Überläufer ist klar, daß die Effizienz der erfolgreichen Suche auch gleichzeitig die Effizienz der Entferne-Operation ist, und daß das Einfügen gerade so effizient ist wie die erfolglose Suche.

Ein Belegungsfaktor von mehr als 1 ist möglich; d.h., selbst wenn die zu verwaltende Datenmenge mehr als vorgesehen wächst, so arbeitet das Verfahren noch korrekt.

Effizienz

Anzahl bei der Suche betrachteter Einträge	separate Verkettung		direkte Verkettung	
	erfolgreich	erfolglos	erfolgreich	erfolglos
$\alpha = 0.50$	1.250	1.110	1.250	0.50
0.90	1.450	1.307	1.450	0.90
0.95	1.475	1.337	1.475	0.95
1.00	1.500	1.368	1.500	1.00

Abb. 4. Effizienz der Verkettung der Überläufer (Anzahl der durchschnittlichen

Die entscheidenden Nachteile der Methoden der Verkettung der Überläufer sind der Speicherplatzbedarf für die Zeiger und die Tatsache, daß selbst dann Platz für Überläufer außerhalb der Hashtabelle benötigt wird, wenn in der Hashtabelle noch viele Plätze frei sind.

- Die gesamte Auslastung und Ausnutzung der eigentlichen Hashtabelle kann niedrig sein trotz hoher Gesamtbelegung.

Offene Hashverfahren

Im Unterschied zur Verkettung der Überläufer außerhalb der Hashtabelle versucht man bei offenen Hashverfahren, Überläufer in der Hashtabelle unterzubringen.

- Wenn also beim Versuch, den Schlüssel k in die Hashtabelle an Position h(k) einzutragen, fest-gestellt wird, daß t[h(k)] bereits belegt ist, so muß man nach einer festen Regel einen anderen, nicht belegten Platz (eine offene Stelle) finden, an dem man k unterbringen kann.

- Da man von vornherein nicht wissen kann, welche Plätze belegt sein werden und welche nicht, definiert man für jeden Schlüssel eine Reihenfolge, in der alle Speicher-plätze, und zwar einer nach dem anderen, betrachtet werden. Sobald ein betrachteter Platz frei ist, wird der Schlüssel dort gespeichert.

    - Die Folge der zu betrachtenden Spei-cherplätze für einen Schlüssel nennt man die Sondierungsfolge zu diesem Schlüssel.
    - Die Folge wird i.A. iterativ/rekursiv berechnet.

Offene Hashverfahren

Die Hauptaufgabe ist nun das Finden eine geeigneten und effizienten Sondierungsfolge und deren Berechnung.

- Natürlich ist das Entfernen von Schlüsseln bei all diesen Verfahren problematisch.
  - Ein bereits in der Hashtabelle vorhandener Schlüssel k1 versperrt ja einem neu einzufügenden Schlüssel k2 im allgemeinen einen Platz, den k2 gemäß seiner Sondierungsfolge betrachtet.
  - Der neue Schlüssel k2 weicht also auf einen anderen Platz (später in der Sondierungsfolge) aus.
  - Wird nun k1 entfernt, so kann k2 nicht wiedergefunden werden, weil der leergewordene Platz von k1 in der Sondierungsfolge von k2 vor dem aktuellen Platz von k2 auftritt.
  - In diesem Fall wird k1 bei den meisten Verfahren dann auch nicht wirklich entfernt, sondern lediglich als entfernt markiert.
  - Wird ein neuer Schlüssel eingefügt, so wird der Platz von k1 als frei angesehen; wird ein Schlüssel gesucht, so wird der Platz von k1 als belegt angesehen.

Offene Hashverfahren

Sei s(j,k) eine Funktion von j und k so, daß (h(k)-s(j,k)) mod m für j = 0,1,..,m-1, eine Sondierungsfolge bildet, d.h. eine Permutation aller Hashadressen.

- Es sei stets noch mindestens ein Platz in der Hashtabelle frei.

- **Suchen** nach Schlüssel k: Beginne mit Hashadresse i = h (k). Solange k nicht in t[i] gespeichert ist und t[i] nicht frei ist, suche weiter bei i = (h(k)-s(j,k)) mod m, für aufsteigende Werte von j. Falls t[i] belegt ist, wurde k gefunden; sonst war die Suche erfolglos.
- **Einfügen** eines Schlüssels k: Wir nehmen an, daß k nicht schon in t vorkommt (das kann durch eine Suche festgestellt werden). Beginne mit Hashadresse i = h(k). Solange t[i] belegt ist, mache weiter bei i = (h(k)-s(j,k)) mod m, für steigende Werte von j. Trage k bei t[i] ein.
- **Entfernen** eines Schlüssels k: Suche nach Schlüssel k. Verläuft die Suche erfolgreich und ist i die Adresse, an der k gefunden wird, dann markiere t[i] als entfernt; sonst kommt k nicht in t vor und kann auch nicht entfernt werden.

Offene Hashverfahren

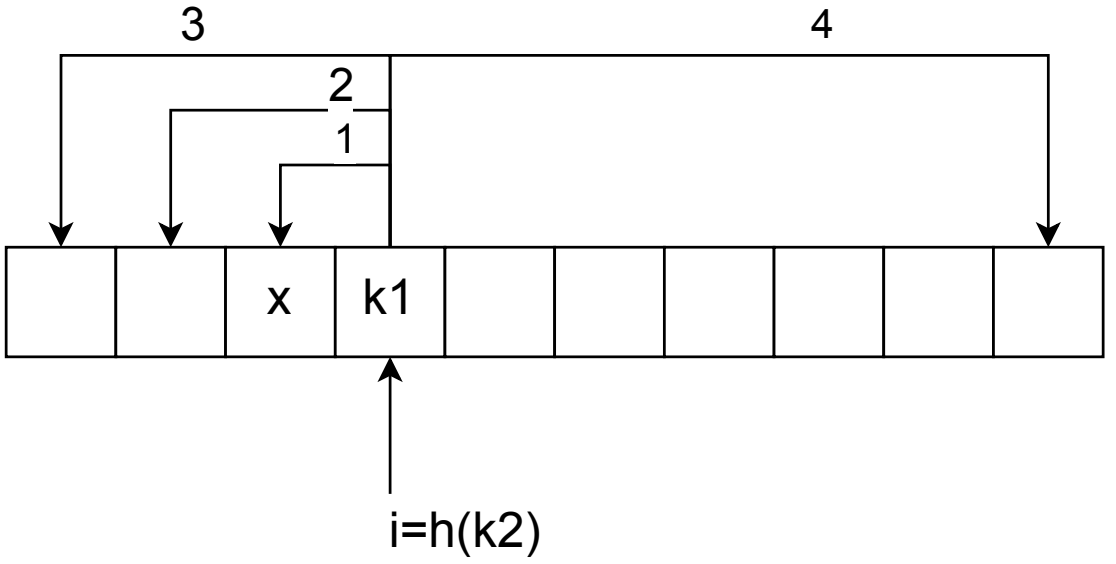

Abb. 5. Prinzip des Sondierens

Lineares Sondieren

$$h(k), h(k)-1, h(k)-2, .., 0, m-1, .., h(k)+1$$
$$s(j,k) = j$$

Anzahl betrachteter Einträge	lineares Sondieren	
	erfolgreich	erfolglos
$\alpha = 0.50$	1.5	2.5
0.90	5.5	50.5
0.95	10.5	200.5
1.00	—	—

Abb. 6. Effizienz beim linearen Sondieren (α ist Belegungsfaktor)

Lineares Sondieren

```
N=7; H=vector(N,-1)
function hash(k,N) { return k % N }
function probe(H,N,i) {
  i=i-1; n=N-1
  while(n && H[i]!=-1) {
    n--
    i=(i-1) % N
  }
  return n==0?-1:i
}
function search(H,N,k) {
  i=hash(k,N)
  if (H[i]==k) return i;
  i=i-1; n=N-1
  while(n && H[i]!=k && H[i]!=-1) {
    n--
    i=(i-1) % N
  }
  return n==0?-1:i
}
function collision(H,N,k) {
  i=hash(k,N)
  return H[i]!=-1
}
```

Alg. 1. Lineares Sondieren

Lineares Sondieren

Quadratisches Sondieren

$$h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, ..$$

$$s(j,k) = \lfloor \frac{j}{2} \rfloor^2 (-1)^j$$

Anzahl betrachteter Einträge	quadratisches Sondieren	
	erfolgreich	erfolglos
$\alpha = 0.50$	1.44	2.19
0.90	2.85	11.40
0.95	3.52	22.05
1.00	—	—

Abb. 7. Effizienz beim qudratischen Sondieren (α ist Belegungsfaktor)

Double Hashing

Die Effizienz des uniformen Sondierens wird bereits annähernd erreicht, wenn man statt einer polynomiellen Permutation für die Sondierungsfolge eine zweite Hashfunktion verwendet; die gewählte Sondierungsfolge für Schlüssel k ist

$$h(k), h(k) - h'(k), h(k) - 2h'(k), .., h(k) - (m-1)h'(k)$$
$$s(j,k) = jh'(k)$$

- Dabei muß h'(k) so gewählt werden, daß für alle Schlüssel k die Sondierungsfolge eine Permutation der Hashadressen bildet.
  - Das bedeutet, daß h'(k) ≠ 0 sein muß und m nicht teilen darf.
  - Wählen wir m als Primzahl, dann gilt dies sicher für jedes h'(k) und für alle k; diese Wahl von m ist ja auch günstig für die Divisions-Rest-Methode bei der Hashfunktion h.
  - Wählt man h'(k) abhängig von h(k), so werden manche (oder gar alle) Synonyme die gleiche Sondierungsfolge haben; eine gewisse sekundäre Häufung ist die Folge.
  - Das kann man vermeiden, wenn man h'(k) von h(k) unabhängig wählt.

Vertiefende Literatur

- Algorithmen und Datenstrukturen, Thomas Ottmann, Peter Widmayer, s. 169 ff., Kap. 4 hashverfahren