Hashtabeller
Weiss, kapitel 5

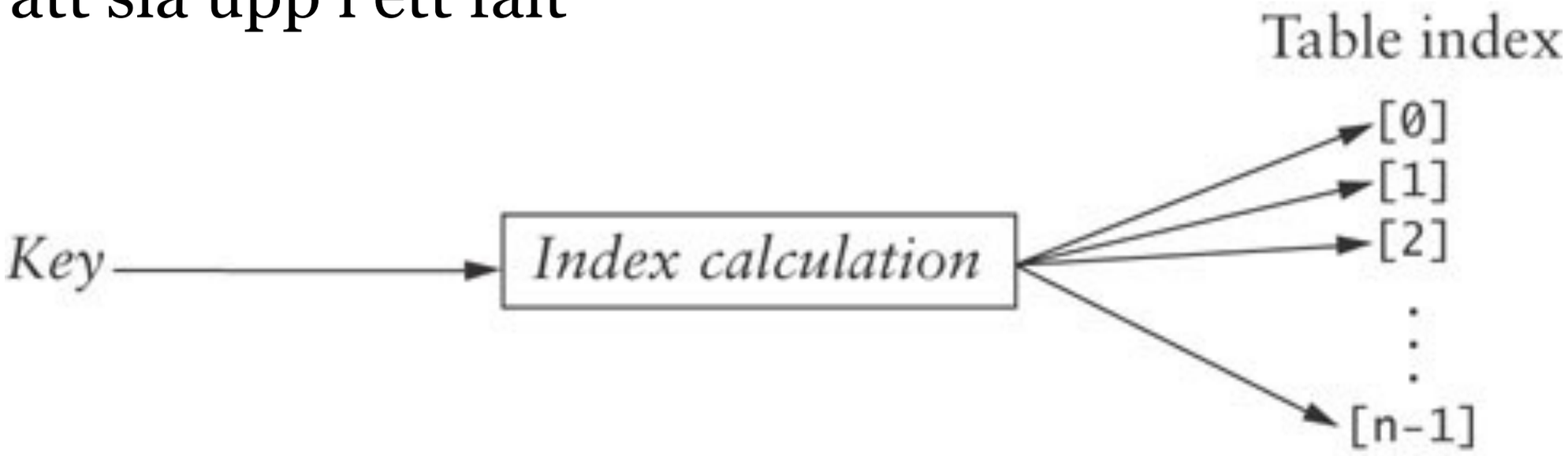
Peter Ljunglöf

Hashtabeller

En hashtabell låter oss slå upp ett värde efter en nyckel istället för en position

Om vi definierar hashtabellen korrekt så blir uppslagning, insättning och borttagning konstanta, O(1), i amorterad tid

Grundidén är att transformera nyckeln till ett heltal (dess hashkod), som sedan används
för att slå upp i ett fält



Hashkoder och tabellindex [5.2]

Antag att vi ska lagra statistik över vilka bokstäver som förekommer i en text.

Om texten bara innehåller ASCII-värden (de första 128 Unicode-tecknen), så kan vi använda ett fält med storlek 128 och låta ASCII-koden vara positionen.

Hur gör vi om vi vill kunna använda alla 1 114 112 Unicode-tecken?

Vi kanske kan använda en tabell med 200 tecken och beräkna index såhär:

```
int index = unicode % 200
```

Hashkoder och tabellindex [5.2]

Antag att vi ska lagra statistik över vilka bokstäver som förekommer i en text.

Om texten bara innehåller ASCII-värden (de första 128 Unicode-tecknen), så kan vi använda ett fält med storlek 128 och låta ASCII-koden vara positionen.

Hur gör vi om vi vill kunna använda alla 1 114 112 Unicode-tecken?

Vi kanske kan använda en tabell med 200 tecken och beräkna index såhär:

```
int index = unicode % 200
```

...	...
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
...	...

Hashkoder och kollisioner

Säg att vi använder en tabell med 200 celler och beräknar:

```
int index = unicode % 200
```

Hur kommer då följande text att kodas?

- *. . . hasta mañana (o el próximo año). . .*

Givet följande Unicode-värden:

Hexadecimal	Decimal	Character	Name
0x0029	41	)	right parenthesis
0x00F1	241	ñ	small letter n with tilde

så blir indexen för bokstäverna ’ñ’ och ’)’ bägge 41

- då får vi en kollision – för att kunna implementera hashtabeller måste vi hantera kollisioner

Hur man genererar hashkoder

I de flesta tillämpningar kommer en nyckel att vara en sträng, eller något ännu mer avancerat

- antalet möjliga nycklar är mycket mycket fler än tabellstorleken
- vi behöver kunna generera hashkoder som ger så få kollisioner som möjligt
- målet är att få en jämn fördelning mellan alla tabellceller

Naiva algoritmer kan ge många kollisioner:

- att summera Unicode-värdena för varje tecken i en sträng ger samma hashkod för ”stram” och ”smart”

Java-metoden .hashCode()

String.hashCode() använder följande formel:

- $c_0 \cdot 31^{n-1} + c_1 \cdot 31^{n-2} + \ldots + c_{n-2} \cdot 31 + c_{n-1}$
  vilket kan beräknas enligt Horners regel:
  $(\ldots((c_0 \cdot 31 + c_1) \cdot 31 + c_1) \cdot \ldots + c_{n-2}) \cdot 31 + c_{n-1}$
- c_k är Unicode-värdet för tecken nr k i strängen, och n är strängens längd
- hashkoden är ett 32-bitars heltal (int), så den trunkeras automatiskt
- 31 är ett primtal, och primtal genererar relativt få kollisioner
- kursboken använder 37 istället (fig. 5.4)

Egenskaper hos hashkoder

Nödvändig egenskap för hashfunktioner:

- $a = b \implies \text{hash}(a) = \text{hash}(b)$

  `a.equals(b)` $\implies$ `a.hashCode() == b.hashCode()`

Följande är dock *ej* nödvändigt:

- $\text{hash}(a) = \text{hash}(b) \implies a = b$

  `a.hashCode() == b.hashCode()` $\implies$ `a.equals(b)`

Önskvärd egenskap för hashfunktioner:

- uniform fördelning, dvs få kollisioner

Beräkna tabellindex [fig. 5.7]

För att få index i hashtabellen (`theLists`) så tar vi resten vid division med storleken på tabellen.

```
private int myhash(AnyType x) {
    int hashVal = x.hashCode();
    hashVal %= theLists.length;
    if (hashVal < 0)
        hashVal += theLists.length;
    return hashVal;
}
```

Eftersom hashkoden är ett `int` kan det vara negativt, och då gör vi det positivt genom att lägga till tabellstorleken.

Två sorters hashtabeller

Det finns två huvudsätt att organisera hashtabeller.

I en hashtabell med *”kedjning” (chaining)* pekar varje tabellcell på en länkad lista med nyckel-värde-par:

- om tre nycklar ger samma hashkod, så har den motsvarande länkade listan tre element
- för att söka efter en viss nyckel så får man gå igenom den länkade listan ett element i taget

I en hashtabell med *öppen adressering* används *”probing”* för att hämta element:

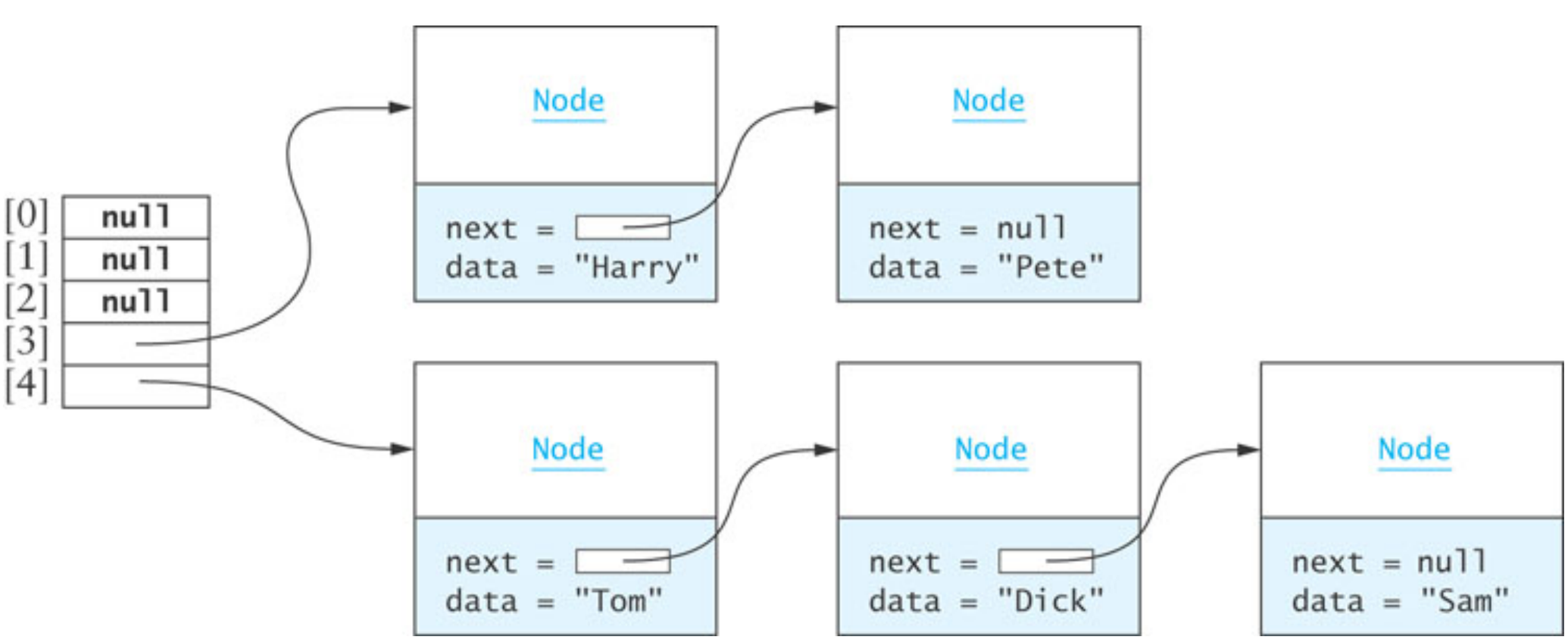
- om tabellcellen för en viss nyckel redan är upptagen med en annan nyckel, så flyttar vi oss till nästa tabellcell
- vi fortsätter flytta oss vidare ända tills vi hittar rätt nyckel eller en tom cell
- detta förutsätter att tabellen aldrig blir full

Chaining

[5.3]

Varje tabellcell pekar på en länkad lista som innehåller alla element med samma hashkod.

Listan är oordnad.



.contains(x) [fig. 5.10]

Vi gör en linjär sökning i den länkade listan för elementets hashkod.

```
public boolean contains(AnyType x) {
    List<AnyType> whichList = theLists[myhash(x)];
    return whichList.contains(x);
}
```

.insert(x) [fig. 5.10]

Vi lägger bara till `x` ifall det inte redan finns i listan.

```
public void insert(AnyType x) {
    List<AnyType> whichList = theLists[myhash(x)];
    if (!whichList.contains(x)) {
        whichList.add(x);
        currentSize++;
        if (currentSize > theLists.length)
            rehash();
    }
}
```

Om tabellen blir för stor måste vi förstora den (`rehash`).

Vi kan använda en annan load factor om vi vill:

`(++currentSize > theLists.length × maxLoadFactor)`

Omhashning – förstora tabellen

En hashtabell med chaining kan innehålla många fler element än storleken på tabellen.

- ju fler element vi lägger till desto längre blir listorna
- listorna växer linjärt med antalet insättningar
- sökning i hashtabellen får linjär komplexitet

Vi måste alltså dubblera tabellen när vi får för många element.

- ungefär som ni gjorde för `DynamicArrayList`
- det går inte att direktkopiera värdena från den gamla tabellen, eftersom hashkoderna kan ändras radikalt
- vi måste gå igenom alla värden i tabellen, ett efter ett, och beräkna en ny hashkod

Mer om detta senare…

.remove(x) [fig. 5.10]

Att ta bort element är omvänt mot att lägga till.

```
public void remove(AnyType x) {
    List<AnyType> whichList = theLists[myhash(x)];
    if (whichList.contains(x)) {
        whichList.remove(x);
        currentSize--;
    }
}
```

Egentligen bör vi även ta bort `whichList` om den blir tom, annars kan effektiviteten bli lidande.

Vi kan också välja att lägga in en omhashning om load factor blir för litet.

Öppen adressering [5.4]

Nackdelen med chaining är att vi måste ha en extra datastruktur (länkade listor), vilket tar mycket extra utrymme i anspråk.

Ett alternativ att lösa konflikter på är att pröva alternativa celler i tabellen ända tills vi kommer till en passande cell.

Mer formellt så prövar vi cellerna $h_0(x)$, $h_1(x)$, ..., där

- $h_i(x)$ = (hash(x) + $f(i)$) % TableSize
- för någon *probing*-funktion f
- vi måste se till att tabellen aldrig blir helt full

Det finns olika probing-funktioner:

- t.ex. linjär, kvadratisk och dubbel-hashning

Linjär probing [5.4.1]

Vid probing prövas cellerna $h_0(x)$, $h_1(x)$, ..., där:

- $h_i(x)$ = (hash(x) + $f(i)$) % TableSize

Vid linjär probing så är f linjär:

- $f(i) = i$
- $h_i(x)$ = (hash(x) + i) % TableSize

Teoretiskt så beror antalet probes på load factor:

- $T_{insert}(\lambda) = \frac{1}{2} \cdot (1 + \frac{1}{(1-\lambda)^2})$
- $T_{search}(\lambda) = \frac{1}{2} \cdot (1 + \frac{1}{(1-\lambda)})$

I praktiken bildas lätt kluster som förvärrar saken.

Att ta bort ett element

När man tar bort ett element, så kan man inte sätta tabellcellen till null:

- om vi letar efter ett annat element med samma hashkod, som råkar ligga efter det borttagna elementet, så hittar vi aldrig det

Istället så lagrar vi ett dummy-värde i tabellcellen

- att ta bort element minskar inte fyllnadsgraden (load factor)

Fördelar/nackdelar

Fördelar med chaining:

- man kan lagra fler element i hashtabellen
- man behöver bara leta igenom de nycklar som har samma hashkod
- vid öppen adressering kan nycklar med olika hashkod ligga omlott i tabellen
- det är enklare att lägga till och ta bort element

Nackdelar med chaining:

- tar mer plats, p.g.a. den länkade listan
- varje listnod tar 3× så mycket plats som en tabellcell
- vi måste fortfarande omallokera tabellen då och då, för att bibehålla konstant komplexitet för uppslagning, insättning och borttagning

Öppen adressering vs. chaining

fyllningsgrad, λ (load factor)	antal jämförelser (öppen adressering)	antal jämförelser (chaining)
0 %	1,00	1,00
25 %	1,17	1,13
50 %	1,50	1,25
75 %	2,50	1,38
85 %	3,83	1,43
90 %	5,50	1,45
95 %	10,50	1,48
100 %	—	1,50
200 %	—	2,00
300 %	—	2,50

Öppen adressering vs. chaining

fyllningsgrad, λ (load factor)	antal jämförelser (öppen adressering)	antal jämförelser (chaining)
0 %	1,00	1,00
25 %	1,17	1,13
50 %	1,50	1,25
75 %	2,50	1,38
85 %	3,83	1,43
90 %	5,50	1,45
95 %	10,50	1,48
100 %	—	1,50
200 %	—	2,00
300 %	—	2,50

men: i praktiken tenderar nycklarna i linjär probing att samla sig i kluster, vilket förvärrar saken

Problem med klustring [fig. 5.12]

Linjär probing tenderar att klustra sig,
vilket försämrar effektiviteten avsevärt.

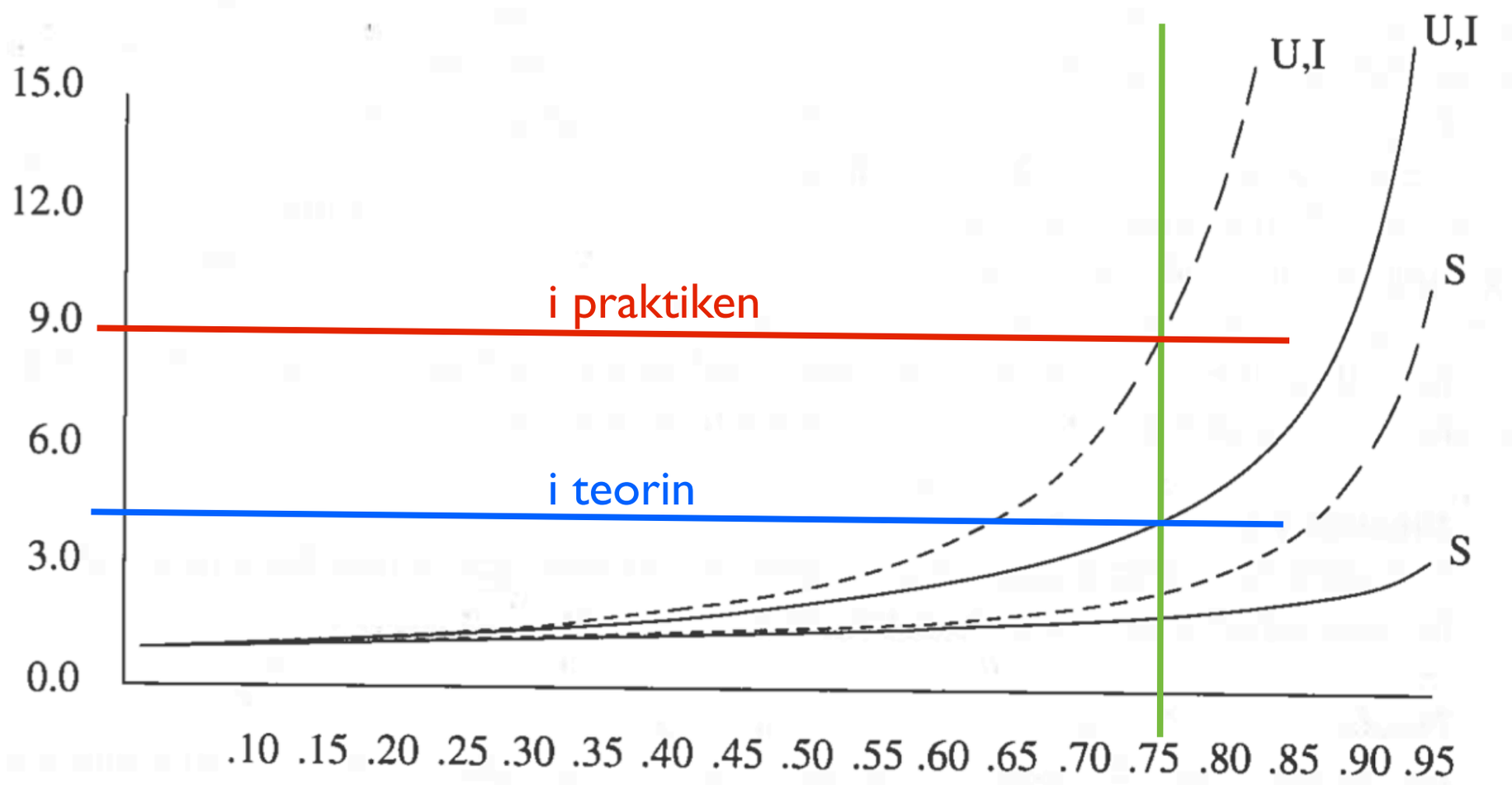


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

Kvadratisk probing [5.4.2]

Vid probing prövas cellerna $h_0(x)$, $h_1(x)$, ..., där:

- $h_i(x)$ = (hash(x) + $f(i)$) % TableSize

Vid kvadratisk probing så är f kvadratisk:

- $f(i) = i^2$
- $h_i(x)$ = (hash(x) + i^2) % TableSize

Detta minskar risken för kluster.

Men: tabellen får aldrig bli mer än halvfull ($\lambda \leq ½$):

- för då finns det risk att man aldrig hittar en ledig cell, om man har otur
- detta kräver också att tabellstorleken är ett primtal!

.contains(x), .isActive(p) [fig. 5.16]

Först letar vi upp cellen där `x` bör kunna ligga, sedan kan vi kontrollera att den inte är tom eller borttagen:

```
public boolean contains(AnyType x) {
    int currentPos = findPos(x);
    return isActive(currentPos);
}
```

En cell är tom om den är null,
den är borttagen om den inte är aktiv:

```
private boolean isActive(int currentPos) {
    return (array[currentPos] != null &&
                array[currentPos].isActive);
}
```

.findPos(x) [fig. 5.16]

Denna hjälpmetod letar upp `x`, eller nästa tomma cell, med hjälp av kvadratisk probing.

```
private int findPos(AnyType x) {
    int offset = 1;
    int currentPos = myhash(x);
    while (!(array[currentPos] == null ||
                array[currentPos].element.equals(x))) {
        currentPos += offset;
        offset += 2;
        if (currentPos ≥ array.length)
            currentPos −= array.length;
    }
    return currentPos;
}
```

Vi utnyttjar $f(i) = f(i-1) + 2i - 1$, för att slippa multiplikationer.

.insert(x) [fig. 5.17]

Vi letar upp en passande cell, och om den är tom så stoppar vi in `x` där.

```
public void insert(AnyType x) {
    int currentPos = findPos(x);
    if (isActive(currentPos))
        return;
    array[currentPos] = new HashEntry<AnyType>(x);
    currentSize++;
    if (currentSize > array.length / 2)
        rehash();
}
```

Eftersom det är kvadratisk probing, så måste vi omhasha så fort load factor $\lambda > ½$.

.remove() [fig. 5.17]

När vi tar bort element så kan vi inte bara göra cellen tom, utan måste välja ett dedikerat objekt.

Eller en flagga (isActive):

```
public void remove(AnyType x) {
    int currentPos = findPos(x);
    if (isActive(currentPos))
        array[currentPos].isActive = false;
}
```

Dubbelhashning [5.4.3]

Kvadratisk probing kan i värsta fall också ge kluster (men det är i extrema undantagsfall).

- dubbelhashning ger ännu mindre klustring

Vid probing prövas cellerna $h_0(x)$, $h_1(x)$, ..., där:

- $h_i(x) = (\text{hash}(x) + f_x(i)) \,\%\, \text{TableSize}$

Vid dubbelhashning så beror f_x på en andra hashfunktion, t.ex.:

- $f_x(i) = i \cdot \text{hash}_2(x)$
- $h_i(x) = (\text{hash}_1(x) + i \cdot \text{hash}_2(x)) \,\%\, \text{TableSize}$

Tyvärr så blir uppslagningarna långsammare...

Omhashning [5.5]

När load factor blir för stor,

- $\lambda > 0.5$ för kvadratisk probing
- $\lambda > 0.5$–0.75 för linjär probing
- $\lambda >$ ca 1.0–3.0 för chaining

så måste vi dubblera tabellstorleken

- det är alltid bra att ha ett primtal som tabellstorlek
- för kvadratisk probing är det dessutom nödvändigt!

Vi kan inte utnyttja gamla tabellindex

- vi måste skapa en ny tabell och stoppa in elementen ett efter ett
- koden för chaining och öppen adressering är lika

.rehash() [fig. 5.22]

Omhashning för en chaining-tabell:

```
private void rehash() {
    List<AnyType> [] oldLists = theLists;

    int newSize = nextPrime(2 × oldLists.length);
    theLists = new List[newSize];

    for (int j = 0; j < theLists.length; j++)
        theLists[j] = new LinkedList<AnyType>();

    currentSize = 0;
    for (List<AnyType> oldList : oldLists)
        for (AnyType item : oldList)
            insert(item);
}
```

.rehash() [fig. 5.22]

Omhashning för en öppen adressering-tabell:

```
private void rehash() {
    HashEntry<AnyType> [] oldArray = array;

    int newSize = nextPrime(2 × oldArray.length);
    array = new HashEntry[newSize];



    currentSize = 0;
    for (HashEntry item : oldArray)
        if (item != null && item.isActive)
            insert(item.element);
}
```

Effektivitet / komplexitet

Insättning, uppslagning och borttagning

- är konstanta, O(1), i *medeltal* (amorterad tid)
- under förutsättning att hashfunktionen är bra (dvs, ger en jämn fördelning av hashkoder)
- dessutom måste vi utöka själva tabellen när fyllnadsgraden blir för hög, även vid chaining

Komplexiteten kan bli linjär, O(*n*), om

- vi har en dålig hashfunktion
- vi aldrig utökar den underliggande tabellen

Läs mer och testa själv!

Wikipedia har mer information om hashtabeller:

http://en.wikipedia.org/wiki/Hash_table

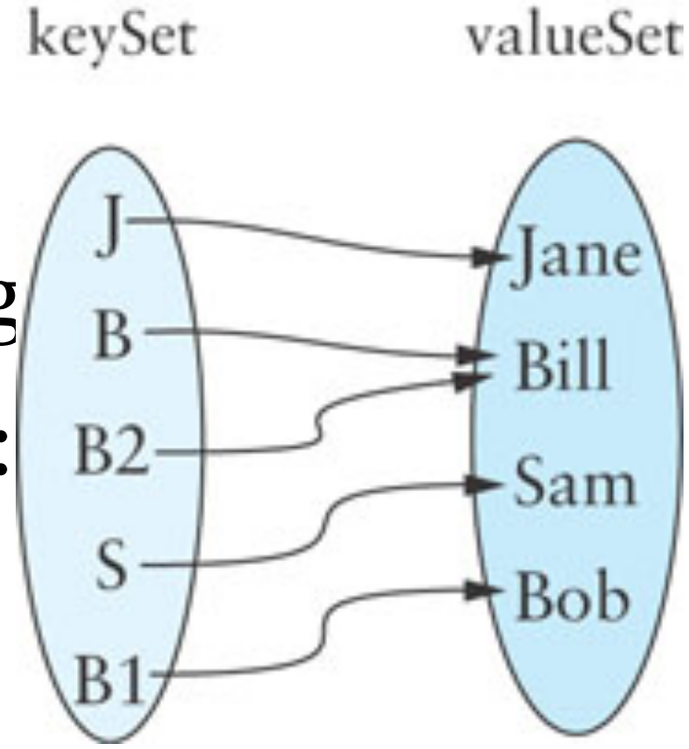
Följande käcka Java-applet låter dig leka med både chaining och linjär probing:

http://www.cse.yorku.ca/~aaw/Hang/hash/Hash.html

Avbildningar: Map<K,V>

En avbildning är ekvivalent med en mängd nyckel-värde-par:

- nycklar måste vara unika
- värden kan förekomma flera gång



Map-gränssnittets viktigaste metoder:

```
interface Map<K,V> {
    public V get(Object key)
    public V put(K key, V value)
}
```

Java har en inbyggd klass `HashMap`, som är en hashtabell-implementation av `Map`

- för tillfället implementerat med chaining

Set<E> vs. List<E>

En mängd kan definieras som en avbildning:

- `Set<E>` = `Map<E,E>`
- där värdet alltid är samma som nyckeln (det tar inget extra minne)

Mängder får endast innehålla unika element:

- metoden `.add(x)` returnerar false om `x` redan finns

Mängder är oordnade:

- mängder har ingen metod `.get(n)`
- du kan iterera genom elementen i en mängd, men ordningen är godtycklig