

F09 - Prioritetskö, Hög, Hashtabell

5DV149 Datastrukturer och algoritmer
Kapitel 13.5, 14.5–14.8

Niclas Börlin
niclas.borlin@cs.umu.se

2024-04-19 Fre

Innehåll

- ▶ Prioritetskö:
 - ▶ Modell
 - ▶ Organisation
 - ▶ Konstruktioner
 - ▶ Listor
 - ▶ *Heap* (Hög)
- ▶ Hashtabell

Prioritetskö

Prioritetskö

- ▶ Modell:
 - ▶ Patienterna på en akutmottagning kommer in i en viss tidsordning men behandlas utifrån en **annan** ordning
- ▶ Organisation:
 - ▶ En **mängd** vars grundmängd är **linjärt ordnad** av en **prioritetsordning**:
 - ▶ Avläsningar och borttagningar görs endast på det element som har **högst prioritet**
 - ▶ Andra mängdoperationer är inte aktuella

Informell specifikation av prioritetskö (1)

```
abstract datatype Pqueue(val, R)
  Empty() → Pqueue(val, R)
  Isempty(p: Pqueue(val, R)) → Bool
  Insert(v: val, p: Pqueue(val, R)) → Pqueue(val, R)
  Inspect-first(p: Pqueue(val, R)) → val
  Delete-first(p: Pqueue(val, R)) → Pqueue(val, R)
  Kill(p: Pqueue(val, R)) → ()
```

- ▶ R är **relationen** för prioritetsordningen
 - ▶ Om t.ex. R är “<” så är “ $a R b$ ” sann om $a < b$

Informell specifikation av prioritetskö (2)

- ▶ `Empty` returnerar en tom prioritetkö
- ▶ `Iempty` returnerar `True` om kön är tom
- ▶ `Insert` stoppar in ett element i kön
- ▶ `Inspect-first` returnerar värdet på elementet med högst prioritet i kön
- ▶ `Delete-first` tar bort elementet med högst prioritet i kön
- ▶ `Kill` lämnar tillbaka alla resurser
- ▶ Vi kommer också att stöta på operationen `Update` som uppdaterar prioritetkön internt

Fråga

- ▶ Hur hanteras element med **samma** prioritet?

Formell specifikation av prioritetskö

OBS! Fel i boken!

Ax 1 lsempty (Empty)

Ax 2 \neg lsempty (Insert (v, p))

Ax 3 Inspect-first (Insert (v, Empty)) = v

Ax 4 Inspect-first (Insert (v1, Insert (v2, p))) =

if v1 R v2

then Inspect-first (Insert (v1, p))

else Inspect-first (Insert (v2, p))

Ax 5 Delete-first (Insert (v, Empty)) = Empty

Ax 6 Delete-first (Insert (v1, Insert (v2, p))) =

if v1 R v2

then Insert (v2, Delete-first (Insert (v1, p)))

else Insert (v1, Delete-first (Insert (v2, p)))

Formell specifikation av prioritetskö

OBS! Fel i boken!

Ax 1 lsempty (Empty)

Ax 2 \neg lsempty (Insert (v, p))

Ax 3 Inspect-first (Insert (v, Empty)) = v

Ax 4 Inspect-first (Insert (v1, Insert (v2, p))) =

if v1 R v2

then Inspect-first (Insert (v1, p))

else Inspect-first (Insert (v2, p))

Ax 5 Delete-first (Insert (v, Empty)) = Empty

Ax 6 Delete-first (Insert (v1, Insert (v2, p))) =

if v1 R v2

then Insert (v2, Delete-first (Insert (v1, p)))

else Insert (v1, Delete-first (Insert (v2, p)))

► Frågor:

- Om R är “<” och två **lika** värden stoppas in, vilket plockas ut **först**?
- Dito om R är “≤”.

Exempel (1)

► För val=heltal, $R=<$, dvs. “ $a R b$ ” är sann om $a < b$:

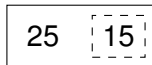
► $p \leftarrow \text{Empty}()$



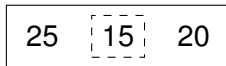
► $p \leftarrow \text{Insert}(25, p)$



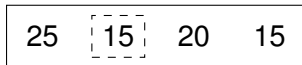
► $p \leftarrow \text{Insert}(15, p)$



► $p \leftarrow \text{Insert}(20, p)$

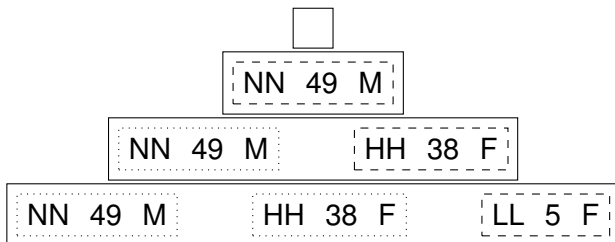


► $p \leftarrow \text{Insert}(15, p)$



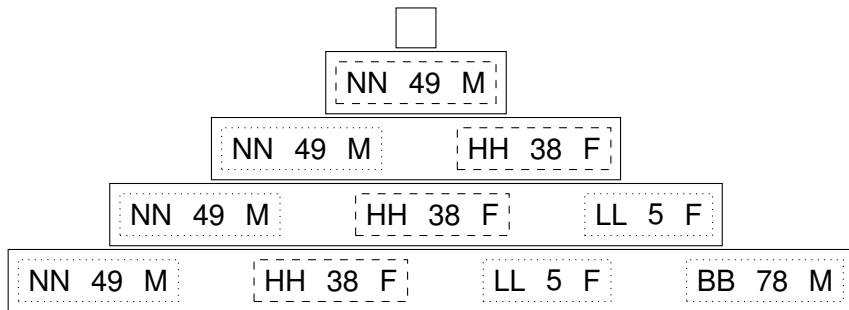
Exempel (2)

- För val=3-tippel med (name, age, sex),
 - `R= a.age < b.age`:



Exempel (3)

- ▶ För val=3-tippel med (name, age, sex)
 - ▶ R= F R M (kvinnor prioriteras före män):



Stack och Kö som specialfall av Prioritetskö

- ▶ Om R är en **strikt partiell ordning**, t.ex. $>$, kommer **lika** element behandlas som en **kö**
- ▶ Om R är **icke-strikt** partiell ordning, t.ex. \geq , behandlas lika element som en **stack**
- ▶ Om R är den **totala relationen**, dvs. sann för alla par av värden blir prioritetskön en **stack**
- ▶ Om R är den **tomma relationen**, dvs. falsk för alla par av värden, blir prioritetskön en **kö**

Fråga

- ▶ Hur lagras elementen **internt** i prioritetskön?

Konstruktioner av Prioritetskö

- ▶ Utgår ofta från konstruktioner av:
 - ▶ Mängd
 - ▶ Lista eller
 - ▶ Hög
- ▶ Notera:
 - ▶ En del konstruktioner av Prioritetskö kan ej upprätthålla ordningen mellan lika element!

Prioritetskö som osorterad lista

► val=heltal, R="<":

Prioritetskö som osorterad lista

► val=heltal, R="<":

► Insert: $O(1)$



Prioritetskö som osorterad lista

► val=heltal, R="<":

► Insert: $O(1)$



Prioritetskö som osorterad lista

► val=heltal, R="<":

► Insert: $O(1)$



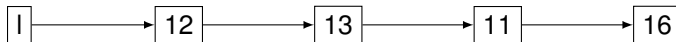
Prioritetskö som osorterad lista

- ▶ val=heltal, R="<":

- ▶ Insert: $O(1)$



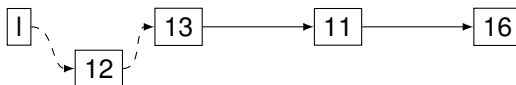
- ▶ Inspect-First, Delete-first: $O(n)$



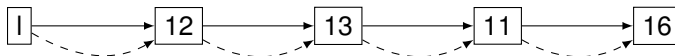
Prioritetskö som osorterad lista

- ▶ val=heltal, R="<":

- ▶ Insert: $O(1)$



- ▶ Inspect-First, Delete-first: $O(n)$



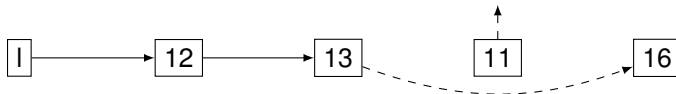
Prioritetskö som osorterad lista

- ▶ val=heltal, R="<":

- ▶ Insert: $O(1)$



- ▶ Inspect-First, Delete-first: $O(n)$



Prioritetskö som sorterad lista

► val=heltal, R="<":

► Insert: $O(n)$



Prioritetskö som sorterad lista

► val=heltal, R="<":

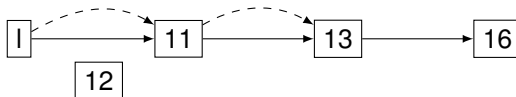
► Insert: $O(n)$



Prioritetskö som sorterad lista

► val=heltal, R="<":

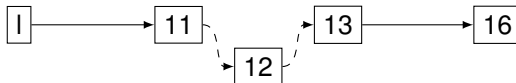
► Insert: $O(n)$



Prioritetskö som sorterad lista

► val=heltal, R="<":

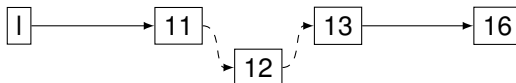
► Insert: $O(n)$



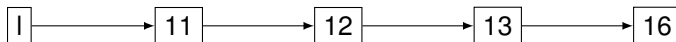
Prioritetskö som sorterad lista

- ▶ val=heltal, R="<":

- ▶ Insert: $O(n)$



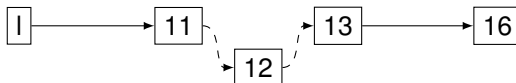
- ▶ Inspect-first, Delete-first: $O(1)$



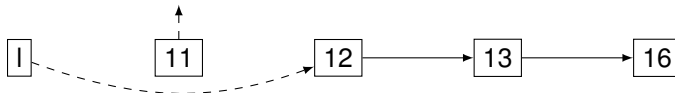
Prioritetskö som sorterad lista

- ▶ val=heltal, R="<":

- ▶ Insert: $O(n)$



- ▶ Inspect-first, Delete-first: $O(1)$



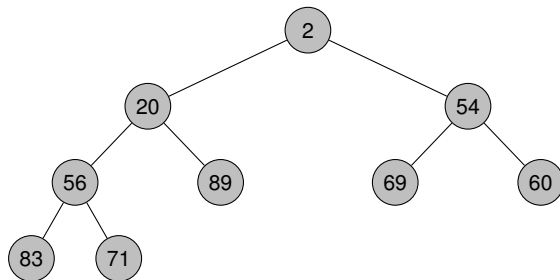
Hög (*Heap*)

Hög (*heap*), informellt

- ▶ En Hög (**heap**) är ett **partiellt sorterat binärt träd**
- ▶ I en Hög så ligger det viktigaste elementet **överst**
 - ▶ Det gäller **rekursivt**
 - ▶ Varje delträd är också en Hög

Hög, formellt

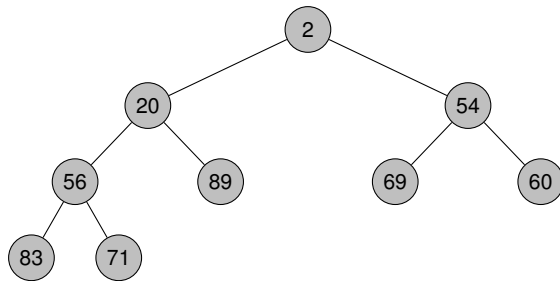
- ▶ Ett binärt träd **är en Hög** eller har **hög-egenskapen** för en relation R om och endast om:
 - ▶ Trädet är sorterat så att etiketterna för alla föräldra-barn-par uppfyller $p R c$, där p är föräldraetiketten och c är barnetiketten
- ▶ Exempel: Är följande träd en Heap med $R <$?



- ▶ Insättningar och borttagningar blir **effektiva** om dom görs så att trädet hålls **komplett**

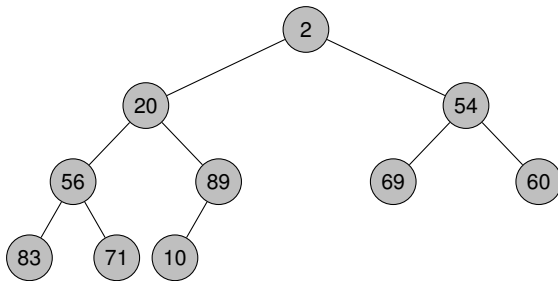
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



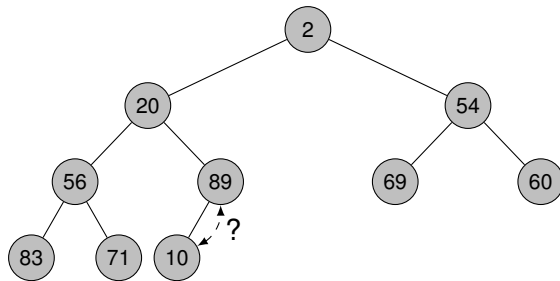
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



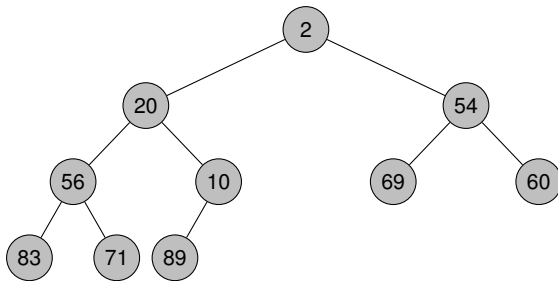
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



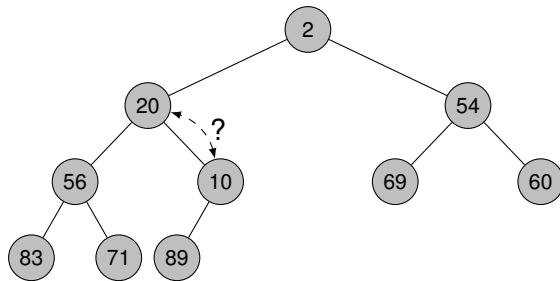
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



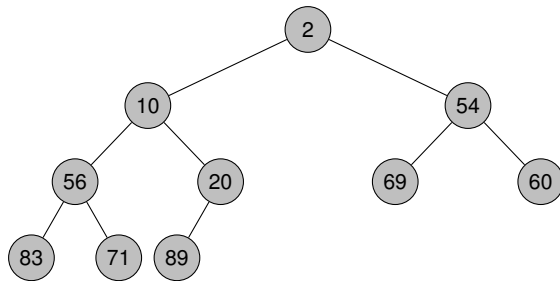
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



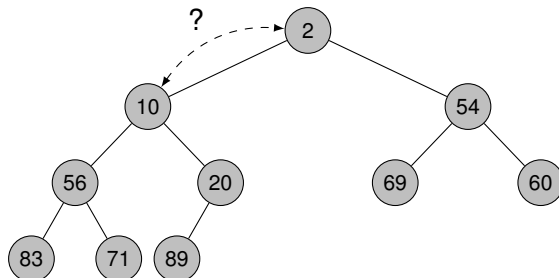
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



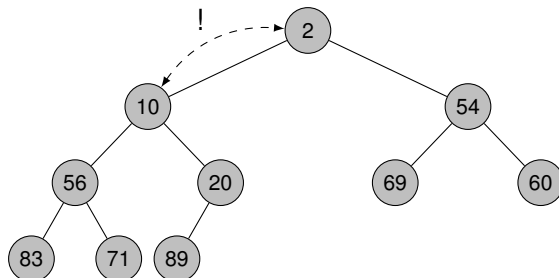
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



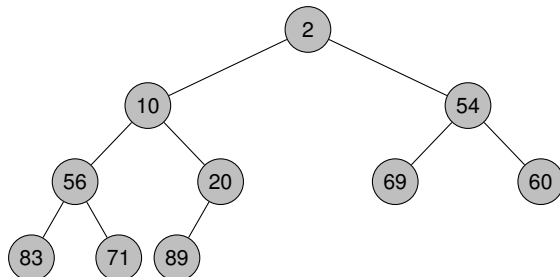
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
 2. **Sortera om** grenen tills trädet är en Hög
- Exempel: Sortera in 10:



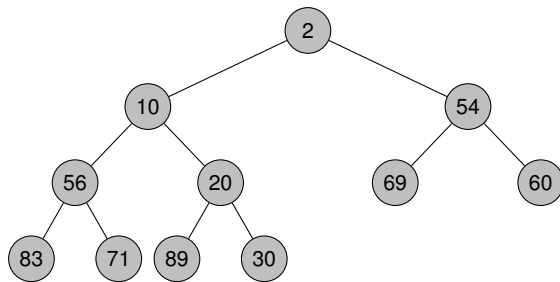
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



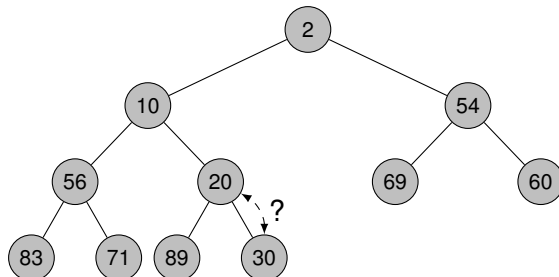
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



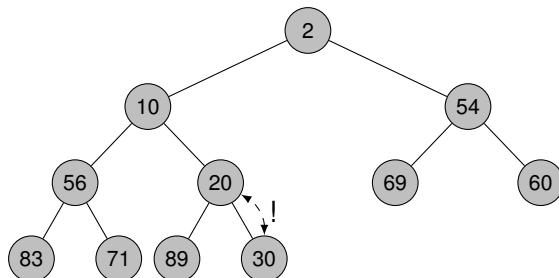
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



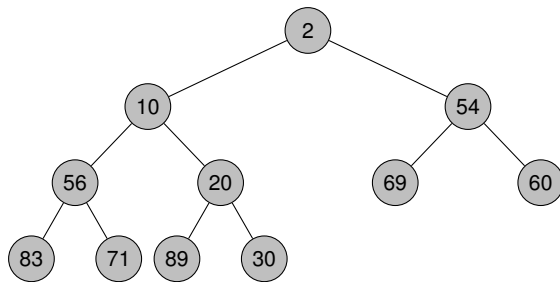
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



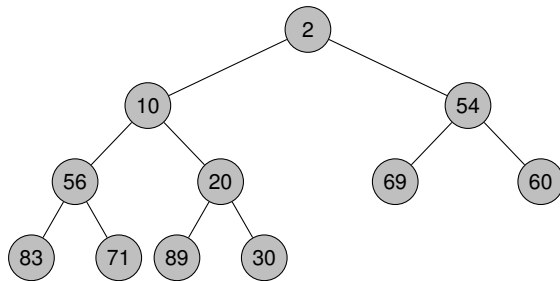
Heap — Algoritm för insättning

1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



Heap — Algoritm för insättning

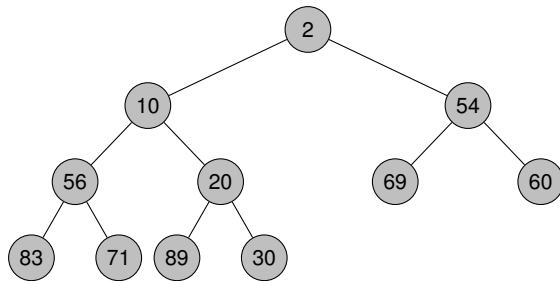
1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



- ▶ Komplexitet för insättning av **ett** element i en Hög med n element?

Heap — Algoritm för insättning

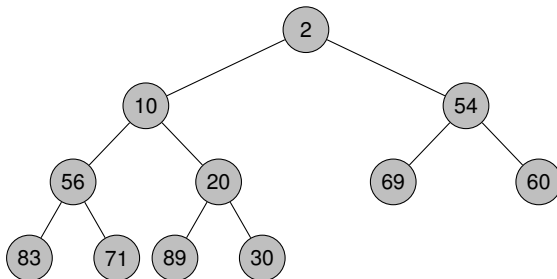
1. Sätt in det nya elementet på den **första lediga platsen**
2. **Sortera om** grenen tills trädet är en Hög
 - ▶ Exempel: Sortera in 10:
 - ▶ Exempel: Sortera in 30:



- ▶ Komplexitet för insättning av **ett** element i en Hög med n element?
 - ▶ $O(\log n)$

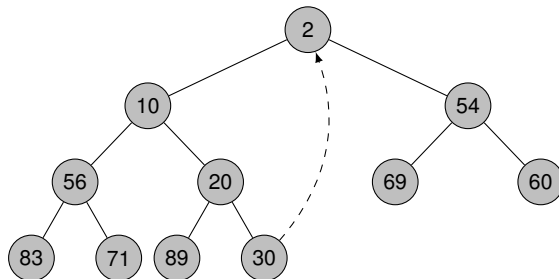
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



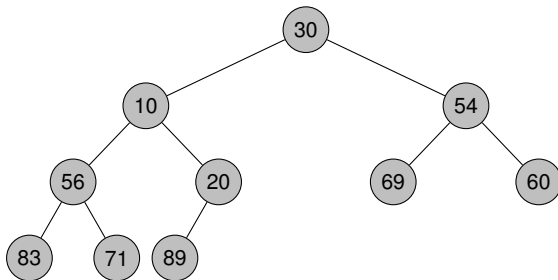
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



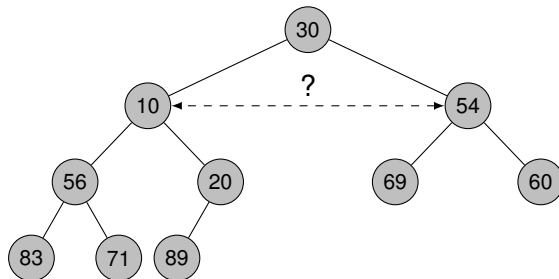
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



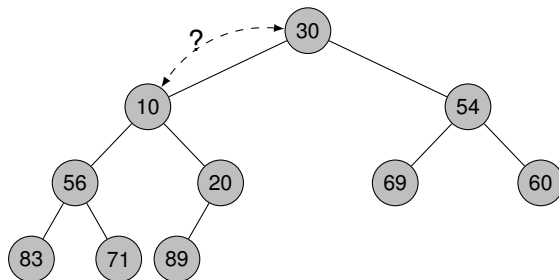
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



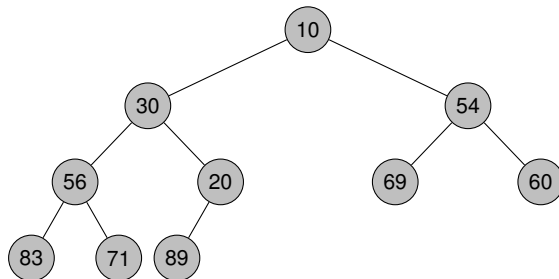
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



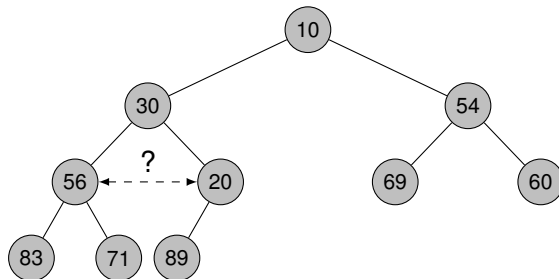
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



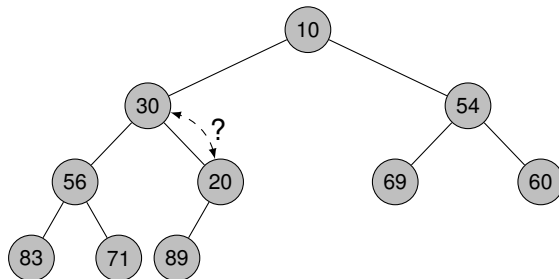
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



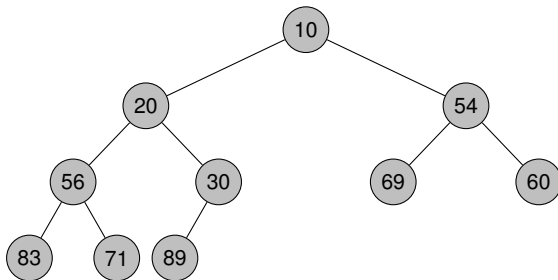
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



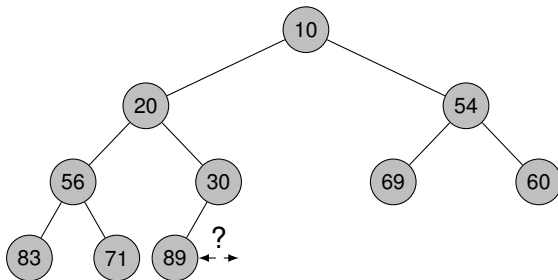
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



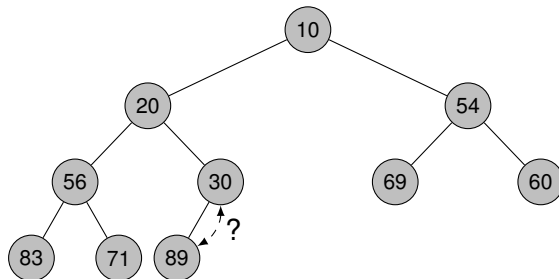
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



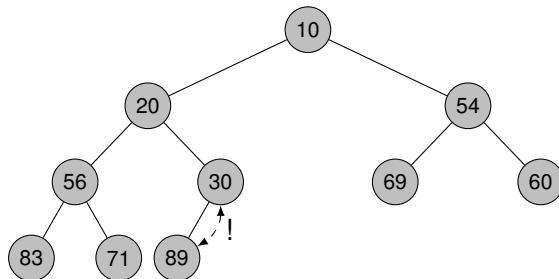
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



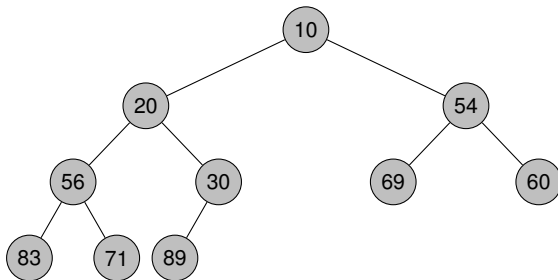
Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



Heap — Algoritm för borttagning

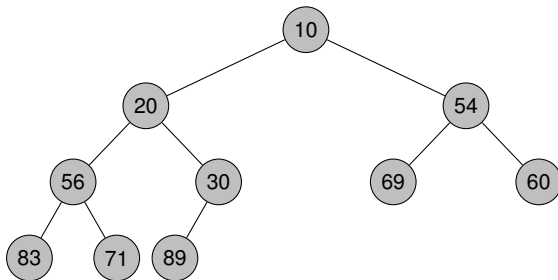
1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



- Komplexitet för borttagning av ett element i en Hög med n element?

Heap — Algoritm för borttagning

1. Ta bort **toppelementet**
 2. Flytta **sista** elementet till toppen
 3. Om nödvändigt,
 - 3.1 Byt ut toppelementet mot det **minsta** av dess barn
 - 3.2 **Fortsätt nedåt** i den påverkade grenen
- Exempel: Remove-first:



- Komplexitet för borttagning av ett element i en Hög med n element?
- $O(\log n)$

Komplexitet för olika konstruktioner av Prioritetskö

	Insättning	Avläsning	Borttagning	Går att göra stabil?
Lista	$O(1)$	$O(n)$	$O(n)$	Ja
Sorterad lista	$O(n)$	$O(1)$	$O(1)$	Ja
Hög	$O(\log n)$	$O(1)$	$O(\log n)$	Nej

Tillämpningar

- ▶ Operativsystem som fördelar jobb mellan olika processer
- ▶ Enkelt sätt att sortera något:
 - ▶ Stoppa in allt i en heap och plocka ut det igen — **heapsort**
 - ▶ Heapsort har komplexitet $O(n \log n)$ och är **instabil**
- ▶ Hjälpmedel vid traversering av **graf**:
 - ▶ Jfr **stack** och **kö** används vid traversering av **träd**

Blank

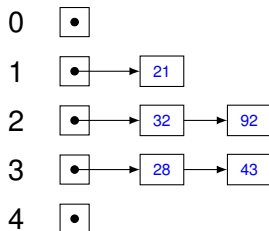
Hashtabell

Långsam sökning i Tabell

- ▶ Problem: Sökningen i en generell Tabell är $O(n)$
- ▶ Vi vill hitta ett **snabbare** sätt att söka

Principskiss Hashtabell

► Öppen hashtabell



► Sluten hashtabell

0	127
1	21
2	32
3	43
4	92
5	127
6	127
7	127
8	28
9	127

Tabell som Fält

- ▶ Om vi kan konstruera tabellen med ett **Fält** blir operationerna *Lookup*, *Insert*, *Remove* $O(1)$

Tabell-funktion	Fält-funktioner
<i>Lookup</i>	<i>Has-value</i> och <i>Inspect-value</i>
<i>Insert</i>	<i>Set-value</i>
<i>Remove</i>	<i>Set-value</i>

- ▶ En **teoretisk** begränsning är att **nyckeltypen** måste gå att använda som **index**
- ▶ En **praktisk** begränsning är att **grundmängden** för nycklarna kan bli **stor**
 - ▶ Ex. nyckelmängden 32-bitars heltal kräver ett Fält med 4 miljarder element

Hashtabell

- ▶ En **Hashtabell** är en variant på tabell som har följande egenskaper
 1. I princip lika **generell** som Tabell
 - ▶ Lite större krav på indextypen
 2. I princip lika **snabb** som Fält
 - ▶ Alla operationer går att göra i $O(1)$ tid, med vissa begränsningar
 3. Kräver **mycket mindre minne** än Fält

Hashtabell, krav på nyckeltypen

- ▶ Precis som för Tabell så kräver vi att **likhet** är definierat för nyckeltypen
- ▶ **Dessutom** kräver vi att det finns en speciell **nyckel-hashfunktion** implementerad för nyckeltypen
- ▶ Önskade egenskaper för nyckel-hashfunktionen:
 - ▶ Nyckel-hashfunktionen är definierad för **alla** objekt ***k*** av nyckeltypen
 - ▶ Nyckel-hash-funktionen tar ett objekt ***k*** av nyckeltypen och returnerar ett **heltal**
 - ▶ Nyckel-hash-funktionen bör vara **snabb** att beräkna
 - ▶ Nyckel-hash-värdena för **olika** nycklar bör vara **olika**
- ▶ Begränsning
 - ▶ Vi har i princip ingen begränsning i **storlek** på heltalet
 - ▶ I praktiken brukar en fysisk datatyp bli begränsningen, t.ex. 32-bitars heltal

Gränsyta för Hashtabell

```
abstract datatype Hashtable(arg, val)
  Empty(kh: function(arg)) → Hashtable(arg, val)
  Insert(k: arg, v: val, t: Hashtable(arg, val))
    → Hashtable(arg, val)
  Isempty(t: Hashtable(arg, val)) → Bool
  Lookup(k: arg, t: Hashtable(arg, val)) → (Bool, val)
  Remove(k: arg, t: Hashtable(arg, val))
    → Hashtable(arg, val)
  Kill(t: Hashtable(arg, val)) → ()
```

Exempel på nyckel-hashfunktion (1)

- För indextypen Sträng så är det vanligt att iterera över alla element i strängen, t.ex.

```
1  Algorithm String-hash(s: String)
2  // Compute a hash value that depends on all characters in the string
3  seed ← 131 // Magic number
4  hash ← 0
5  for i from 0 to length(s) - 1 do
6    hash ← (hash * seed) + char-to-int(lowercase(Inspect-value(s, i)))
7  return hash
```

- Notera att hash-värdet kan bli stort!
 - Strängen "Jan" får hash-värdet 1831883

Exempel på nyckel-hashfunktion (2)

- ▶ För en Post bör alla relevanta fält påverka hashvärdet
- ▶ För en Post med fälten `item_number` (Heltal) och `serial_number` (Sträng) skulle nyckel-hashfunktionen kunna vara

```
1 Algorithm Record-hash(r: Record)  
2 return r.item_number * String-hash(r.serial_number)
```

- ▶ Posten (1412, "LE74D") får då hash-värdet
 $1412 \cdot 32033999426 = 45232007189512$

Notera

- ▶ Det var vanligt att motiveringen till hashtabeller är att det är möjligt att använda **strängar** som nycklar
- ▶ Jag vill poängtera att det går att använda **vilken datatyp som helst** som nyckel, så länge en nyckel-hashfunktion finns definierad

Tabell-hashfunktion

- ▶ Alla hashtabeller är konstruerade med något sorts **Fält**
- ▶ Nyckel-hash-värdena går att använda som index i fältet, men är i praktiken **för stora**
- ▶ För att lösa det kommer vi att använda en **tabell-hashfunktion**
- ▶ Tabell-hashfunktionen avbildar en **stor** indextyp A på en **mindre** indextyp B, t.ex.
 - ▶ A=32-bitars heltal, B=8-bitars heltal
 - ▶ A=heltal i intervallet 0–99, B=heltal i intervallet 0–6
- ▶ Vi lagrar sedan våra tabellvärden i ett Fält med indextyp B

Tabell-hashfunktion, önskade egenskaper

- ▶ En hashfunktion $h(a)$ bör följande egenskaper:
 1. Funktionen kan avbilda **alla** element a i A på **något** element $b = h(a)$ i B
 2. De avbildade elementen $b = h(a)$ bör ha en bra **spridning** för de förväntade värdena i A
 3. Funktionen är **snabb** att beräkna för alla a
- ▶ För exemplet postnummer med intervallen $A=10000-99999$, $B=0-99$ skulle hashfunktionen

$$h(x) = \lfloor x/1000 \rfloor$$

ha egenskap 1 och 3 men inte 2:

- ▶ Inget värde avbildas på 0–9
- ▶ Fler värden avbildas troligtvis på 11 än 98 (fler postnummer 11xxx än 98xxx)

Operatörn mod

- ▶ Den överlägset vanligaste tabell-hashfunktionen använder operatörn **mod** som beräknar **heltalsrest** vid division
 - ▶ Operatörn är snabb och har **bra spridningsegenskaper** på många indata
 - ▶ För Heltal $a, n > 0$ så avbildar

$$h(a) = a \bmod n,$$

alla heltalen a på heltalen $[0, 1, \dots, n - 1]$

- ▶ Ex.

$0 \bmod 4 = 0,$	$4 \bmod 4 = 0,$
$1 \bmod 4 = 1,$	$5 \bmod 4 = 1,$
$2 \bmod 4 = 2,$	$6 \bmod 4 = 2,$
$3 \bmod 4 = 3,$	$7 \bmod 4 = 3$

- ▶ Om a är ett nyckel-hash-värde och n är storleken på fältet så kan $h(a) = a \bmod n$ användas som index i hashtabellen (fältet)

Summering (1)

- Nyckel-hashfunktionen

$$a = n(k) \in A$$

ger oss **generalitet** för nyckeltypen men riskerar att ge stora hash-värden

- Tabell-hashfunktionen

$$h(a) = h(n(k)) \in B$$

reducerar nyckel-hashvärdena till ett litet intervall och ger oss lågt **minnesutnyttjande**

Summering (2)

- ▶ Återstår att visa att funktionerna `Lookup`, `Insert`, `Remove` går att implementera i $O(1)$ tid
- ▶ Alla funktionerna har behov av att hitta en plats (`index`) i hashtabellen (fältet) givet en `nyckel` k
 - ▶ Nyckeln k **duger inte som index**, då k typiskt inte är av indextypen för fältet
 - ▶ Nyckel-hash-värdet $n(k)$ är ett heltal, men kan vara **för stort**
 - ▶ Tabell-hash-värdet $h(n(k))$ går att använda som index, men flera nycklar kan avbildas på **samma hashvärde**
 - ▶ Detta kallas för en **kollision**
 - ▶ Vi kommer att använda oss av en **sökalgoritm** som hanterar kollisioner

Blank

Kollisionshantering

Förenkling

- ▶ När vi diskuterar kollisionshantering kommer vi att använda nyckeltypen Heltal dvs. nyckel-hashfunktionen är **identitetsfunktionen**

```
1  Algorithm Int-hash(i: Int)
2  // The hash value for an integer is the value itself
3  return i
```

- ▶ Vi kommer också att använda **nyckelvärde** som **tabellvärde**
 - ▶ I princip implementera ett Lexikon
- ▶ Senare kommer vi att titta på exempel för mer **generella** nyckeltyper och tabeller

Kollisioner

- ▶ En **kollision** är när två nycklar avbildas på **samma** hash-värde
 - ▶ En bra hashfunktion förväntas generera "få" kollisioner
 - ▶ Kollisioner går ej att undvika **helt**
- ▶ Exempelvis skulle

$$h(x) = x \mod 10$$

avbilda nyckelvärdena 89 och 59 på index 9

- ▶ Kollisioner kan hanteras med
 1. **Öppen** hashning
 2. **Sluten** hashning
 - 2.1 **Linjär** teknik
 - 2.2 **Kvadratisk** teknik

Blank

Öppen hashning

Öppen hashning

- ▶ Vi använder en k -Vektor av Lista av nyckel-värden och tabell-hash-funktionen

$$h(x) = x \mod k$$

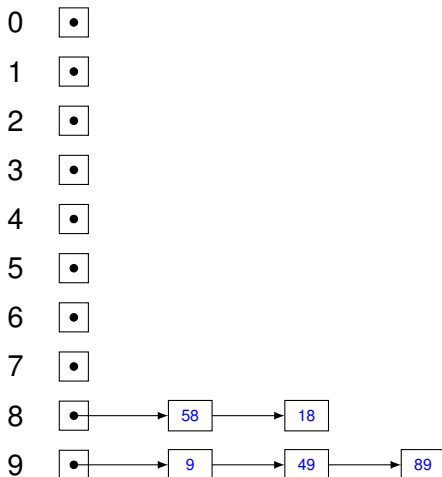
- ▶ Vi får en **dynamisk** tabell, ingen begränsning på antalet element
- ▶ Alla element med hashvärde x hamnar i listan med index $h(x)$

Insert i öppen Hashtabell

```
1  Algorithm Hash-table-open-insert(x: Hashvalue, t: Hashtable)
2  // Insert in a open hashtable. Returns the updated hash table.
3  // Does not check whether the value is already in the table.
4  //
5  // Values are stored in lists. The lists are stored in a zero-indexed
6  // vector t.v.
7
8  // Compute the size of the hash table.
9  size ← High(t.v) + 1
10 // Map hash value to the table size
11 h ← x mod size
12
13 // Insert in the list at index h
14 l ← Array-inspect-value(t.v, h)
15 // Insert at the first position in the list
16 // Note that this may produce duplicates in the list
17 l ← List-insert(l, List-first(l), x)
18 t.v ← Array-set-value(t.v, h, l)
19
20 return t
```

Öppen hashning, exempel

- ▶ Sätt in talen 89, 18, 49, 58, 9 i en Hashtabell med 10 platser:



Lookup i öppen Hashtabell

```
1  Algorithm Hash-table-open-lookup(x: Hashvalue, t: Hashtable)
2  // Lookup from a open hashtable.
3  //
4  // Values are stored in lists. The lists are stored in a zero-indexed vector t.v.
5  //
6  // If the element is found, return True and the stored element value. Otherwise,
7  // return False.
8
9  // Compute the size of the hash table.
10 size ← High(t.v) + 1
11
12 // Map hash value to the table size
13 h ← x mod size
14
15 // Lookup in the list at index h
16 l ← Array-inspect-value(t.v, h)
17
18 // Iterate over the list
19 p ← List-first(l)
20 while not List-isend(l, p) do
21   // Inspect the element
22   v ← List-inspect(l, p)
23   if v = x then
24     // We found the requested element.
25     return (True, v)
26   else
27     // Advance in the list
28     p ← List-next(l, p)
29
30 // We've checked all elements in the list.
31 // The element is not there.
32 return (False, None)
```


Remove i öppen Hashtabell

```
1  Algorithm Hash-table-open-remove(x: Hashvalue, t: Hashtable)
2  // Remove from a open hashtable. Returns the updated hash table.
3  //
4  // Values are stored in lists. The lists are stored in a zero-indexed
5  // vector t.v of size TABLE_SIZE.
6
7  // Compute the size of the hash table.
8  size ← High(t.v) + 1
9
10 // Map hash value to the table size
11 h ← x mod size
12
13 // Extract the list at index h
14 l ← Array-inspect-value(t.v, h)
15
16 // Iterate over the list
17 p ← List-first(l)
18 while not List-isend(l, p) do
19   if List-inspect(l, p) = x then
20     // Remove the current element. The returned position is the
21     // position AFTER the removed. Continue to traverse the list
22     // since we may have duplicates.
23     (l, p) ← List-remove(l, p)
24   else
25     // Advance in the list
26     p ← List-next(l, p)
27
28 // Remove in list complete, now re-insert the list in the array
29 t.v ← Array-set-value(t.v, h, l)
30
31 return t
```

Öppen hashning, summering

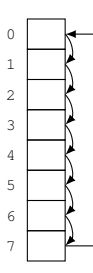
- ▶ Värstafallskomplexitet:
 - ▶ $O(n)$ för alla operationer (alla element i samma lista), där n är antalet insatta element
- ▶ Medelfallskomplexitet:
 - ▶ Insättning och misslyckad sökning blir n/k
 - ▶ Lyckad sökning blir ungefär $n/2k$
- ▶ Tumregel: **Maximalt $2k$ element** bör sättas in

Blank

Sluten hashning

Sluten hashning, sökning

- ▶ Vid sluten hashning används en **cirkulär**, noll-baserad **vektor** (fält) för att lagra datat
 - ▶ **Statisk** tabell med fast antal platser
 - ▶ Index i en tabell av storlek n räknas **modulo n**
 - ▶ Det första elementet följer efter det sista
- ▶ Vi kommer att behöva reservera två värden som **markörer** i tabellen
 - ▶ Värdet `HASH_EMPTY` kommer att användas som en markör att platsen är **ledig**
 - ▶ Värdet `HASH_REMOVED` kommer att användas som en markör att platsen är ledig, men att ett värde har **tagis bort** från platsen någon gång
- ▶ Markörvärdena får inte vara giltiga tabellvärden
 - ▶ Om vi vill lagra värden 0–99 i hashtabellen kan vi t.ex. välja
 - ▶ `HASH_EMPTY` = 100, `HASH_REMOVED` = 101 eller
 - ▶ `HASH_EMPTY` = -1, `HASH_REMOVED` = -2



Sluten hashning, sökning med linjär teknik

- ▶ **Sökning** (Lookup) efter ett element k börjar på index $h(k)$ och fortsätter eventuellt **framåt**
 - ▶ Om värdet inte påträffats **före nästa lediga plats** så finns det inte i tabellen
- ▶ Vid sökning med **linjär** teknik testas följande index i sekvens:

$$\begin{array}{l} (h(k) + 0) \mod n \\ (h(k) + 1) \mod n \\ (h(k) + 2) \mod n \\ (h(k) + \vdots) \mod n \\ (h(k) + n - 1) \mod n \end{array}$$

Intern sökfunktion som använder linjär teknik

```
1  Algorithm Hash-table-closed-internal-lookup-linear(x: Hashvalue, t: Hashtable)
2  // Internal lookup function used by the hash table functions.
3  // Look up the hash value x in the hash table t. Use linear collision handling.
4  //
5  // If a match is found, return True and the index where the match was found.
6  // If an empty slot is found, return False and the index of the empty slot.
7  // If no match nor an empty slot was found, return False and no index.
8
9  // The values are stored in a zero-based vector t.v.
10 // Compute the size of the hash table.
11 size ← High(t.v) + 1
12 // Map hash value to the table size
13 h ← x mod size
14
15 // Check each available index
16 for i from 0 to size - 1 do
17     // Compute index with linear collision handling
18     j ← (h + i) mod size
19
20     // Inspect the element at index j in the internal vector
21     v ← Array-inspect-value(t.v, j)
22
23     if v = x then
24         // We found the value, return True and the index
25         return (True, j)
26
27     if e = HASH_EMPTY then
28         // We found an empty slot; the value cannot be in the table
29         return (False, j)
30
31 // We have checked all indices without finding the value
32 return (False, None)
```

Sökning i sluten Hashtabell med linjär teknik

► Allt jobb görs av den interna sökfunktionen

```
1  Algorithm Hash-table-closed-lookup(x: Hashvalue, t: Hashtable)
2  // Lookup in a closed hashtable using linear collision handling.
3  // Returns True/False and optionally the found value.
4
5  // All work is done by the internal search function
6  (b, j) ← Hash-table-closed-internal-lookup-linear(x, t)
7
8  if b then
9    return (b, Array-inspect-value(t.v, j))
10 else
11   return (False, None)
```


Sluten hashning, insättning

- ▶ Vid **insättning** av ett element görs först en **sökning**
 - ▶ Om sökningen hittar värdet så **ersätts** värdet i tabellen
 - ▶ Om sökningen **inte** hittar värdet så sätts värdet in på den första **lediga** platsen
- ▶ Vilken plats ett element hamnar på beror alltså på två saker:
 1. Dess hashvärde $h(n(k))$ och
 2. vilka värden som redan finns i tabellen

Insättning i sluten Hashtabell med linjär teknik

```
1  Algorithm Hash-table-closed-insert(x: Hashvalue, t: Hashtable)
2  // Insert in a closed hashtable using linear collision handling.
3  // Returns the updated hash table.
4
5  // First check if the value is already in the table.
6  (b, j) ← Hash-table-closed-internal-lookup-linear(x, t)
7
8  if b then
9      // Overwrite the value
10     t.v ← Array-set-value(t.v, j, x)
11 else
12     // Otherwise, search for HASH_REMOVED
13     (b, j) ← Hash-table-closed-internal-lookup-linear(HASH_REMOVED, t)
14
15     if j = None then
16         // Hash table is full
17         Error("Hash table is full")
18     else
19         // Value at index j is either HASH_EMPTY or HASH_REMOVED
20         // In either case, we can put the value there
21         t.v ← Array-set-value(t.v, j, x)
22
23 return t
```

Sluten hashning, linjär teknik, insättning (1)

- ▶ Talen $\{0, \dots, 125\}$ ska lagras i en hashtabell av storlek 10
 - ▶ Vi har alltså hashfunktionen

$$h(x) = x \bmod 10$$

- ▶ Låt 127 betyda “ledig” (`HASH_EMPTY`) och 126 betyda “borttagen” (`HASH_REMOVED`)

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

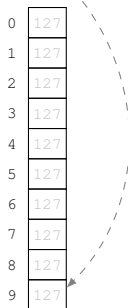
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	127

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

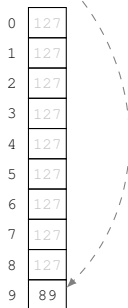


Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)



0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

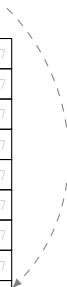
$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

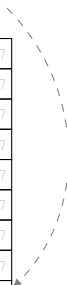
$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

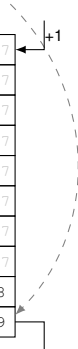
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

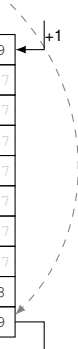
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)


0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

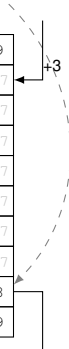
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

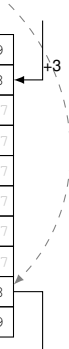
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

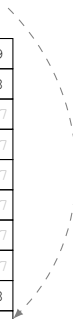
0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

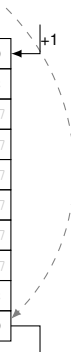
0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)


0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)


0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)


0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, insättning (2)

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	58
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

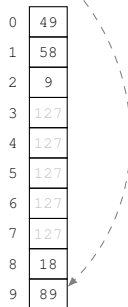
Lookup (49) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

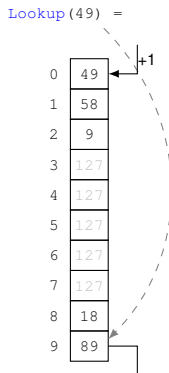
Lookup (49) =



0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

`Lookup(49) = True`

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

Lookup(49) = True

Lookup(79) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

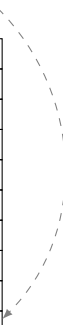
- Sök efter 49, 79, respektive 9:

Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



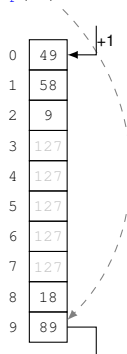
Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

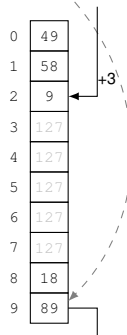
- Sök efter 49, 79, respektive 9:

Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, sökning

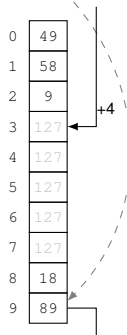
- Sök efter 49, 79, respektive 9:

Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

`Lookup(49) = True`

`Lookup(79) = False`

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

Lookup(49) = True

Lookup(79) = False

Lookup(9) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

Lookup(49) = True

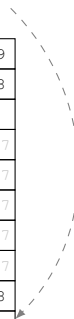
0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

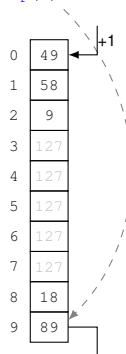
Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

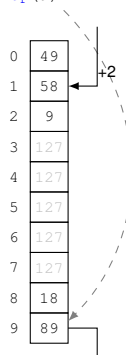
Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

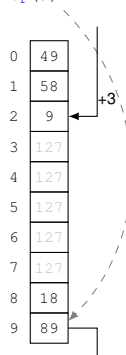
Lookup(49) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:

Lookup(49) = True

Lookup(79) = False

Lookup(9) = True

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, borttagning

- ▶ Vid **borttagning** av ett element används samma sökalgoritm
 - ▶ Om värdet hittas i tabellen kan platsen inte lämnas **tom** (**HASH_EMPTY**) — då kan senare sökningar misslyckas
 - ▶ I stället sätts markören **HASH_REMOVED** in i tabellen

Borttagning i sluten Hashtabell med linjär teknik

```
1 Algorithm Hash-table-closed-remove(x: Hashvalue, t: Hashtable)
2 // Remove in a closed hashtable using linear collision handling.
3 // If the value is not found, does nothing.
4
5 // Search for the value
6 (b, j) ← Hash-table-closed-internal-lookup-linear(x, t)
7
8 if b then
9 // Mark the slot with the value HASH_REMOVED
10 t.v ← Array-set-value(t.v, j, HASH_REMOVED)
11
12 return t
```

Sluten hashning, linjär teknik, borttagning

- ▶ Ta bort 49:
 - ▶ Sätt in "borttagen"-markören på 49:ans plats

Före borttagning Efter borttagning

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

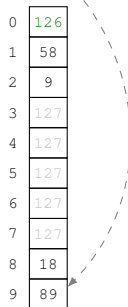
Lookup (49) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

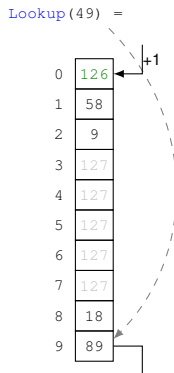
Lookup (49) =



0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49

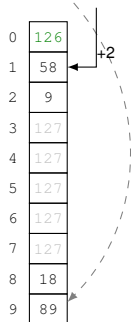
- Sök efter 49, 79, respektive 9:



Sökning efter borttagning av 49

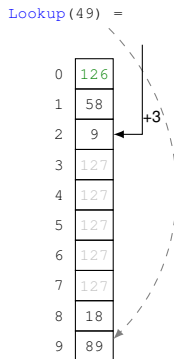
- Sök efter 49, 79, respektive 9:

Lookup (49) =



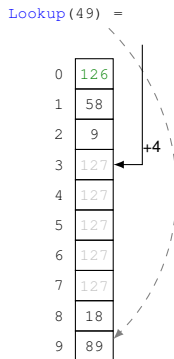
Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

`Lookup(49) = False`

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

`Lookup(49) = False`

`Lookup(79) =`

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49

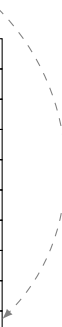
- Sök efter 49, 79, respektive 9:

Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sökning efter borttagning av 49

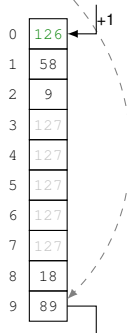
- Sök efter 49, 79, respektive 9:

Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sökning efter borttagning av 49

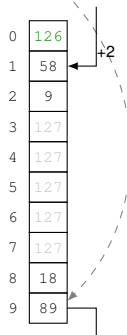
- Sök efter 49, 79, respektive 9:

Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49


- Sök efter 49, 79, respektive 9:

Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

Lookup(49) = False

Lookup(79) = False

Lookup(9) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

Lookup(49) = False

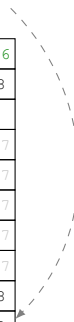
0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

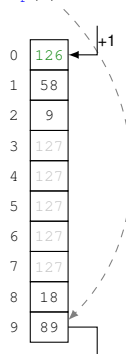
Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

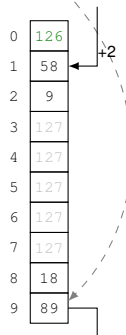
Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =



Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

Lookup(49) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(79) = False

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Lookup(9) =

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

A dashed curved arrow labeled '+3' starts from the value 126 at index 0 and points to the value 9 at index 2.

Sökning efter borttagning av 49

- Sök efter 49, 79, respektive 9:

`Lookup(49) = False`

`Lookup(79) = False`

`Lookup(9) = True`

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

0	126
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, linjär teknik, komplexitet

- ▶ Värstafallskomplexiteten för samtliga operationer är $O(n)$, där n är antalet element som finns insatta i tabellen
- ▶ Är dock ytterst osannolikt
 - ▶ Alla element måste ligga i en följd
- ▶ Under förutsättning att tabellen inte fylls mer än till en "viss del" får man i medeltal $O(1)$ för operationerna
 - ▶ Hur mycket är "till en viss del"?

Fyllnadsgrad

- ▶ En Hashtabells **fyllnadsgrad** (λ) definieras som *kvoten mellan antalet insatta element och antalet platser i tabellen*
- ▶ En tom tabell har $\lambda = 0$ och en full $\lambda = 1$

Hashtabeller, medelkomplexitet

- ▶ Det finns formler för medelantalet platser som måste provas vid olika fyllnadsgrader
- ▶ För en halvfull tabell $\lambda = 0.5$ gäller

Operation	Medelantalet sökningar
Insättning	2.5
Misslyckad sökning	2.5
Lyckad sökning	1.5

- ▶ Slutsats:
 - ▶ Medelkomplexiteten för `Lookup`, `Insert`, `Remove` är $O(1)$ om fyllnadsgraden $\lambda < 1/2$!

Hashtabeller, klustering

- ▶ Linjär teknik ger upphov till **klustering**, dvs. att de upptagna positionerna tenderar att bli "ihopklumpade"

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

- ▶ Låt oss studera ett alternativ till linjär teknik. . .

Sluten hashning, kvadratisk teknik

- Vid sökning med **linjär** teknik testas följande index i sekvens:

$$\begin{array}{l} (h(k) + 0^1) \mod n \\ (h(k) + 1^1) \mod n \\ (h(k) + 2^1) \mod n \\ (h(k) + \vdots) \mod n \\ (h(k) + (n-1)^1) \mod n \end{array}$$

- Vid sökning med **kvadratisk** teknik testas i stället följande index i sekvens:

$$\begin{array}{l} (h(k) + 0^2) \mod n \\ (h(k) + 1^2) \mod n \\ (h(k) + 2^2) \mod n \\ (h(k) + \vdots) \mod n \\ (h(k) + (n-1)^2) \mod n \end{array}$$

Intern sökfunktion som använder kvadratisk teknik

► Endast **en** rad modifierad i pseudokoden!

```
1  Algorithm Hash-table-closed-internal-lookup-quadratic(x: Hashvalue, t: Hashtable)
2  // Internal lookup function used by the hash table functions.
3  // Look up the hash value x in the hash table t. Use quadratic collision handling.
4  //
5  // If a match is found, return True and the index where the match was found.
6  // If an empty slot is found, return False and the index of the empty slot.
7  // If no match nor an empty slot was found, return False and no index.
8
9  // The values are stored in a zero-based vector t.v.
10 // Compute the size of the hash table.
11 size ← High(t.v) + 1
12 // Map hash value to the table size
13 h ← x mod size
14
15 // Check each available index
16 for i from 0 to size - 1 do
17     // Compute index with quadratic collision handling
18     j ← (h + i * i) mod size
19
20     // Inspect the element at index j in the internal vector
21     v ← Array-inspect-value(t.v, j)
22
23     if v = x then
24         // We found the value, return True and the index
25         return (True, j)
26
27     if e = HASH_EMPTY then
28         // We found an empty slot; the value cannot be in the table
29         return (False, j)
30
31 // We have checked all indices without finding the value
32 return (False, None)
```


Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

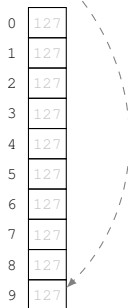
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	127

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

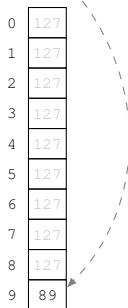


Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)



0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Sluten hashing, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \bmod 10$$

Insert (89)

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

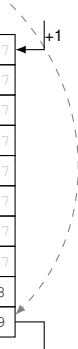
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

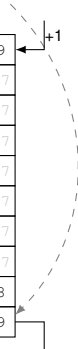
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

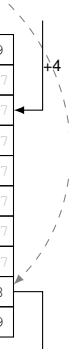
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

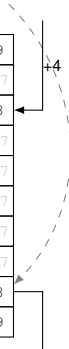
0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

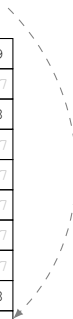
0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)


0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89



Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	127
2	58
3	9
4	127
5	127
6	127
7	127
8	18
9	89

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

$$h(x) = x \mod 10$$

Insert (89)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	127
9	89

Insert (18)

0	127
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (49)

0	49
1	127
2	127
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (58)

0	49
1	127
2	58
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Insert (9)

0	49
1	127
2	58
3	9
4	127
5	127
6	127
7	127
8	18
9	89

Hashtabeller, klustring igen

- Klustringen är ett mindre problem för kvadratisk teknik än för linjär

Linjär

0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Kvadratisk

0	49
1	127
2	58
3	9
4	127
5	127
6	127
7	127
8	18
9	89

- Begränsning för att kvadratisk teknik ska fungera:
 - Längden på tabellen måste vara ett **primtal**
 - Fyllnadsgraden måste vara lägre än 1/2

Summering

- ▶ Medelantal sonderinger för $\lambda = 1/2$:

Operation	linjär teknik	kvadratisk teknik
Insättning	2.5	2.0
Misslyckad sökning	2.5	2.0
Lyckad sökning	1.5	1.4

- ▶ Slutsats:

- ▶ Medelkomplexiteten för `Lookup`, `Insert`, `Remove` är $O(1)$ om fyllnadsgraden $\lambda < 1/2$!
- ▶ Om tabellens storlek är ett **primtal** kan **kvadratisk** teknik användas och är då snabbare än linjär teknik

Mer avancerade hashfunktioner

- ▶ Mer avancerade hashfunktioner kommer från talteori, t.ex.

$$h(x) = ((c_1 x + c_2) \bmod p) \bmod m,$$

- ▶ divisorn p är ett stort primtal $> m$, t.ex. 1048583,
- ▶ konstanterna c_1 och c_2 är heltal > 0 och $< p$.
- ▶ Ännu mer avancerade
 - ▶ md5 (<https://en.wikipedia.org/wiki/MD5>)
 - ▶ SHA-1, SHA-2
(<https://en.wikipedia.org/wiki/Shasum>)
 - ▶ kryptografiska

Tabell som Hashtabell (1)

- ▶ I exemplen har vi hittills använt **heltal** som nycklar och tabellvärden
 - ▶ I princip implementerat ett **Lexikon**
 - ▶ Dessutom har inga exempel innehållit **hash-dubletter** (olika nycklar som genererar samma hash-värde)

Tabell som Hashtabell (2)

- ▶ Följande exempel visar mer realistiska Tabell-operationer
 - ▶ Vi använder en **nyckel-hashfunktion** (`Key-hash-value`) för att beräkna ett hashvärde från ett nyckelvärde
 - ▶ Vi använder **mod** för att trunkera nyckel-hashvärdet så det ryms i Hashtabellen
 - ▶ För att kunna hantera hash-dubletter jämför vi **både** hash-värden **och** nyckel-värden (`Key-compare`) för att hitta en match
 - ▶ Vi lagrar **tabellvärden** i Tabellen
- ▶ Tabellen är konstruerad som en **post** (*record*) med ett enda fält (*field*) \vee som är ett fält (*array*)
 - ▶ Varje **element** som är lagrat i fältet är en **post med tre fält**:
 - key** nyckel-värdet
 - hash** hash-värdet för nyckeln
 - value** tabell-värdet

Intern lookup-funktion

```
1  Algorithm Hash-table-closed-internal-lookup-linear-full(k: Key, t: Hashtable)
2  // Internal lookup function used by the hash table functions.
3  // Look up the key k in the hash table t. Use linear collision handling.
4  // ...
5
6  // Compute the size of the hash table.
7  size ← High(t.v) + 1
8
9  // Use the user-defined hash function to compute the key hash value
10 key-hash ← Key-hash-value(k)
11
12 // Map hash value to the table size
13 h ← key-hash mod size
14
15 // Check each available index
16 for i from 0 to size - 1 do
17   // Compute index with linear collision handling
18   j ← (h + i) mod size
19
20   // Inspect the element at index j in the internal vector
21   v ← Array-inspect-value(t.v, j)
22
23   if key-hash = v.hash then
24     if Key-compare(k, v.key) then
25       // We found the value, return True and the index
26       return (True, j)
27
28   if v.hash = HASH_EMPTY then
29     // We found an empty slot; the value cannot be in the table
30     return (False, j)
31
32 // We have checked all indices without finding the value
33 return (False, None)
```

Exempel, Month-to-days, sluten hashning (1)

Key	Jan	Feb	Mar	Apr	May	Jun
Hash	1831883	1763751	1883370	1679403	1883377	1834503
mod 19	17	0	14	12	2	15

Key	Jul	Aug	Sep	Oct	Nov	Dec
Hash	1834501	1680047	1986858	1917956	1902369	1729430
mod 19	13	10	9	1	13	12

Exempel, Month-to-days, sluten hashning (2)

► Linjär teknik

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Feb 28	Oct 31	May 31							Sep 31	Aug 31		Apr 30	Jul 31	Mar 31	Jun 30	Nov 30	Jan 31	Dec 31

► Kvadratisk teknik

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Feb 28	Oct 31	May 31	Nov 30						Sep 31	Aug 31		Apr 30	Jul 31	Mar 31	Jun 30	Dec 31	Jan 31	

► Jämförelse

	Antal lediga sekvenser	Lyckad sökning	Misslyckad sökning
Linjär	2	$21/12=1.75$	$77/19=4.05$
Kvadratisk	3	$17/12=1.42$	$53/19=2.79$

Exempel, Month-to-days, öppen hashning

Size	Hash Mod	Jan 1831883	Feb 1763751	Mar 1883370	Apr 1679403	May 1883377	Jun 1834503
6		5	3	0	3	1	3

Size	Hash Mod	Jul 1834501	Aug 1680047	Sep 1986858	Oct 1917956	Nov 1902369	Dec 1729430
6		1	5	0	2	3	2

► Öppen hashning, storlek 6

0	Sep	Mar		
1	Jul	May		
2	Dec	Oct		
3	Nov	Jun	Apr	Feb
4				
5	Aug	Jan		

- Lyckad sökning: $22/12 = 1.83$
- Misslyckad sökning: $15/6 = 2.5$