F09 - Hashtabell, Prioritetskö, Hög

5DV149 Datastrukturer och algoritmer
Kapitel 13.5, 14.5–14.8

Niclas Börlin
niclas.borlin@cs.umu.se

2024-02-08 Tor

Innehåll

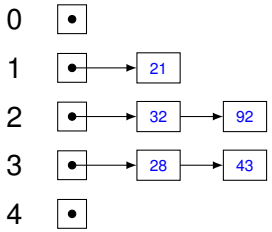
Hashtabell

Långsam sökning i Tabell

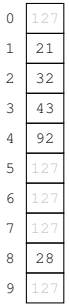
- Problem: Sökningen i en generell Tabell är $O(n)$
- Vi vill hitta ett snabbare sätt att söka

Principskiss Hashtabell

- Öppen hashtabell



- Sluten hashtabell



Tabell som Fält

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

Tabell-funktion	Fält-funktioner
`Lookup`	`Has-value` och `Inspect-value`
`Insert`	`Set-value`
`Remove`	`Set-value`

- 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.

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

```
Algorithm Record-hash(r: Record)
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
- Vi skulle i princip kunna välja att använda nyckel-hash-värdena direkt som index i Fältet
  - Det slösar dock stora mängder minne
- För att slippa skapa onödigt stora Fält 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)

Operatorn mod

- Den överlägset vanligaste tabell-hashfunktionen använder operatorn mod som beräknar heltalsrest vid division
  - Operatorn än snabb och har bra spridningsegenskaper på många indata
  - För Heltal $a, n > 0$ så avbildar

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

alla heltalen a på heltalen $[0, 1, \ldots, n-1]$
  - Ex.

$$\begin{aligned} 0 &\mod 4 = 0, & 4 &\mod 4 = 0, \\ 1 &\mod 4 = 1, & 5 &\mod 4 = 1, \\ 2 &\mod 4 = 2, & 6 &\mod 4 = 2, \\ 3 &\mod 4 = 3, & 7 &\mod 4 = 3 \end{aligned}$$

  - Om a är ett nyckel-hash-värde och n är storleken på fältet så kan $h(a) = a \mod n$ 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

Kollisionshantering

Förenkling

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

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

- Vi kommer också att använda nyckelvärdet 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

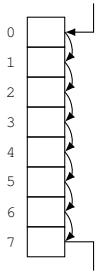
$$h(x) = x \mod 10$$

  avbilda nyckelvärdena 89 och 59 på index 9
- Kollisioner kan hanteras med
  1. Sluten hashning
     - 1.1 Linjär teknik
     - 1.2 Kvadratisk teknik
  2. Öppen hashning

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:

$$
\begin{array}{llll}
(& h(k) & + & 0 &) & \bmod n \\
(& h(k) & + & 1 &) & \bmod n \\
(& h(k) & + & 2 &) & \bmod n \\
 & & & \vdots & & \\
(& h(k) & + & n-1 &) & \bmod n
\end{array}
$$

Sökning i sluten Hashtabell med linjär teknik

```
1  Algorithm Hash-table-lookup-closed-linear-simplified(x: Hashvalue, t: Hashtable)
2  // Lookup in a closed hashtable using linear collision handling.
3  // Returns True/False and optionally the index where x was found.
4  //
5  // Values are stored in a zero-indexed vector t.v of size TABLE_SIZE.
6
7  // Map hash value to the table size
8  h ← x mod TABLE_SIZE
9
10 // Check each available index
11 for i from 0 to TABLE_SIZE - 1 do
12   // Compute index with linear collision handling
13   j ← (h + i) mod TABLE_SIZE
14
15   // Inspect element at index j in the internal vector
16   e ← Array-inspect-value(t.v, j)
17   if e = x then
18     // We found the value
19     return (True, j)
20
21   if e = HASH_EMPTY then
22     // We found an empty slot; the value cannot be in the table
23     return (False, None)
24
25 // We have checked all indices without finding the value
26 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-insert-closed-linear-simplified(x: Hashvalue, t: Hashtable)
2  // Insert in a closed hashtable using linear collision handling.
3  // Returns the updated hash table.
4  //
5  // Values are stored in a zero-indexed vector t.v of size TABLE_SIZE.
6
7  // Map hash value to the table size
8  h ← x mod TABLE_SIZE
9
10 // Check each available index
11 for i from 0 to TABLE_SIZE - 1 do
12   // Compute index with linear collision handling
13   j ← (h + i) mod TABLE_SIZE
14
15   // Inspect element at index j in the internal vector
16   e ← Array-inspect-value(t.v, j)
17   if e = HASH_EMPTY or e = HASH_REMOVED then
18     // We found a free index; insert the value here
19     t.v ← Array-set-value(t.v, j, x)
20     return t
21
22 // We have checked all indices without finding an empty one
23 return Error("Could not find an empty slot")
```


Sluten hashning, linjär teknik, insättning (1)

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

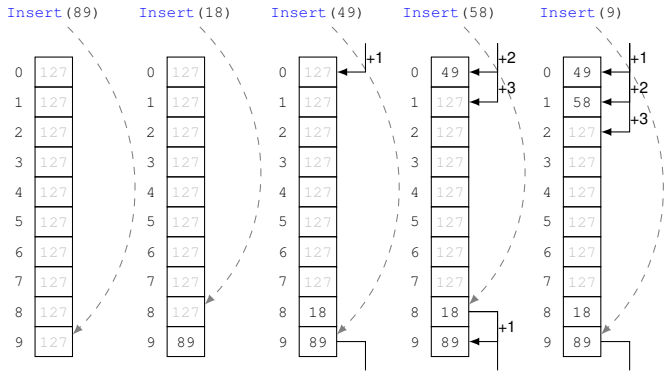
$$h(x) = x \mod 10$$

- Låt `127` betyda "ledig" (`HASH_EMPTY`) och `126` betyda "borttagen" (`HASH_REMOVED`)

Sluten hashning, linjär teknik, insättning (2)

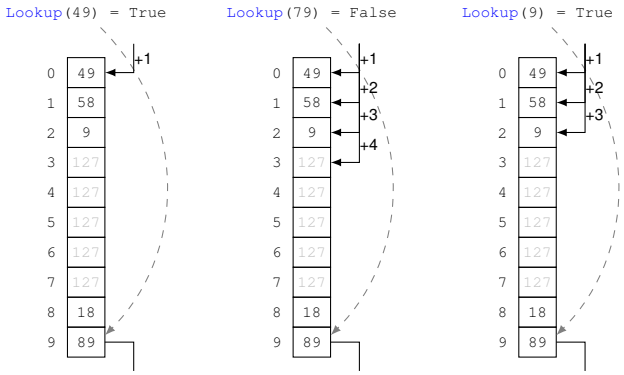
- Sätt in talen $89, 18, 49, 58, 9$ med

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



Sluten hashning, linjär teknik, sökning

- Sök efter 49, 79, respektive 9:



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-remove-closed-linear-simplified(x: Hashvalue, t: Hashtable)
2  // Remove in a closed hashtable using linear collision handling.
3  // Returns the updated hash table.
4  //
5  // Values are stored in a zero-indexed vector t.v of size TABLE_SIZE.
6
7  // Map hash value to the table size
8  h ← x mod TABLE_SIZE
9
10 // Check each available index
11 for i from 0 to TABLE_SIZE - 1 do
12   // Compute index with linear collision handling
13   j ← (h + i) mod TABLE_SIZE
14
15   // Inspect element at index j in the internal vector
16   e ← Array-inspect-value(t.v, j)
17   if e = x then
18     // We found the value; insert the REMOVED value
19     t.v ← Array-set-value(t.v, j, HASH_REMOVED)
20     return t
21
22   if e = HASH_EMPTY then
23     // We found an empty slot; return the unchanged table
24     return t
25
26 // We have checked all indices without finding the value; return the unchanged table
27 return t
```


Sluten hashning, linjär teknik, borttagning

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

Före borttagning

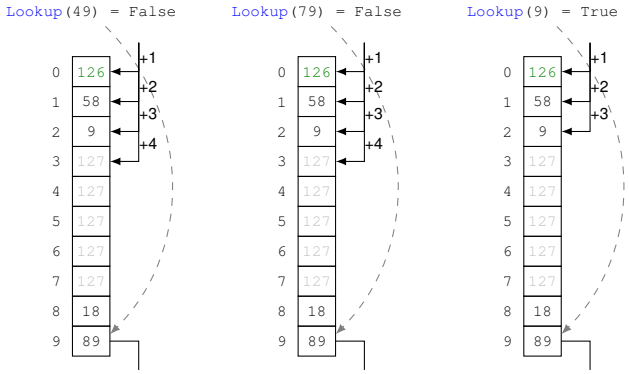
0	49
1	58
2	9
3	127
4	127
5	127
6	127
7	127
8	18
9	89

Efter borttagning

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:



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 prövas 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, klustring

- Linjär teknik ger upphov till klustring, 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:

$$
\begin{aligned}
&(\; h(k) + 0^1 \;) \mod n \\
&(\; h(k) + 1^1 \;) \mod n \\
&(\; h(k) + 2^1 \;) \mod n \\
&(\; h(k) + \vdots \;) \mod n \\
&(\; h(k) + (n-1)^1 \;) \mod n
\end{aligned}
$$

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

$$
\begin{aligned}
&(\; h(k) + 0^2 \;) \mod n \\
&(\; h(k) + 1^2 \;) \mod n \\
&(\; h(k) + 2^2 \;) \mod n \\
&(\; h(k) + \vdots \;) \mod n \\
&(\; h(k) + (n-1)^2 \;) \mod n
\end{aligned}
$$

Sökning i sluten Hashtabell med kvadratisk teknik

- Endast en rad modifierad i pseudokoden!

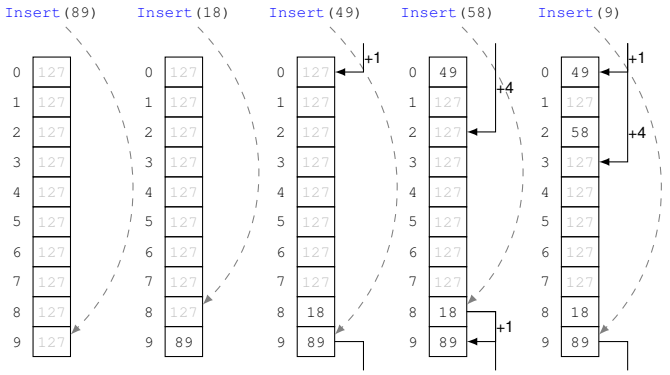
```
1  Algorithm Hash-table-lookup-closed-quadratic-simplified(x: Hashvalue, t: Hashtable)
2  // Lookup in a closed hashtable using quadratic collision handling.
3  // Returns True/False and optionally the index where x was found.
4  //
5  // Values are stored in a zero-indexed vector t.v of size TABLE_SIZE.
6
7  // Map hash value to the table size
8  h ← x mod TABLE_SIZE
9
10 // Check each available index
11 for i from 0 to TABLE_SIZE - 1 do
12   // Compute index with linear collision handling
13   j ← (h + i^2) mod TABLE_SIZE
14
15   // Inspect element at index j in the internal vector
16   e ← Array-inspect-value(t.v, j)
17   if e = x then
18     // We found the value
19     return (True, j)
20
21   if e = HASH_EMPTY then
22     // We found an empty slot; the value cannot be in the table
23     return (False, None)
24
25 // We have checked all indices without finding the value
26 return (False, None)
```

- Dito för `Insert` och `Remove`

Sluten hashning, kvadratisk teknik, insättning

- Sätt in talen 89, 18, 49, 58, 9 med

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



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

Kvadratisk teknik, problem

- Med olyckligt vald hashfunktion och tabellängd finns risk att man inte hittar en ledig plats även om den finns!
- Exempel:
  - Tabellstorlek = 16, hashfunktion $h(x) = x \mod 16$
  - Efter att ha stoppat in elementen 0, 16, 32, och 64, dvs. $\lambda = 1/4$, finns ingen plats för tal med $h(x) = 0$

i	0	1	2	3	4	5	6	7
$h(x) + i^2$	0	1	4	9	16	25	36	49
$h(x) + i^2 \mod 16$	0	1	4	9	0	9	4	1
i	8	9	10	11	12	13	14	15
$h(x) + i^2$	64	81	100	121	144	169	196	225
$h(x) + i^2 \mod 16$	0	1	4	9	0	9	4	1

  - De enda positioner som testas är 0, 1, 4, 9, . . . , som redan är upptagna!

Lösning

- Om kvadratisk teknik används och tabellens storlek är ett primtal så kan ett nytt element alltid stoppas in om $\lambda < 1/2$
  - Ger mindre klustring än linjär hashning
  - 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

Blank

Öppen hashning

Öppen hashning

- I stället för en Vektor av nyckel-värden används en k-Vektor av Lista av nyckel-värden

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

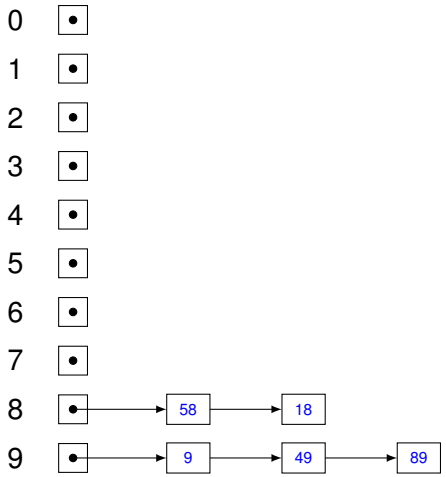
- Vi får en dynamisk tabell, ingen begränsning på antalet element
- Alla element med hashvärde x hamnar i listan med index x

Insert i öppen Hashtabell

```
Algorithm Hash-table-insert-open-simplified(x: Hashvalue, t: Hashtable)
// Insert in a open hashtable. Returns the updated hash table.
// Does not check whether the value is already in the table.
//
// Values are stored in lists. The lists are stored in a zero-indexed
// vector t.v of size TABLE_SIZE.

// Map hash value to the table size
h ← x mod TABLE_SIZE

// Insert in the list at index h
l ← Array-inspect-value(t.v, h)
// Insert at the first position in the list
// Note that this may produce duplicates in the list
l ← List-insert(l, List-first(l), x)
t.v ← Array-set-value(t.v, h, l)

return t
```


Öppen hashning, exempel

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



Lookup i öppen Hashtabell

```
Algorithm Hash-table-lookup-open-simplified(x: Hashvalue, t: Hashtable)
// Lookup from a open hashtable. Returns True/False.
//
// Values are stored in lists. The lists are stored in a zero-indexed
// vector t.v of size TABLE_SIZE.

// Map hash value to the table size
h ← x mod TABLE_SIZE

// Lookup in the list at index h
l ← Array-inspect-value(t.v, h)
// Iterate over the list
p ← List-first(l)
while not List-isend(l, p) do
  if List-inspect(l, p) = x then
    // We found the requested element.
    return True
  else
    // Advance in the list
    p ← List-next(l, p)

// We've checked all elements in the list.
// The element is not there.

return False
```


Remove i öppen Hashtabell

```
Algorithm Hash-table-remove(x: Hashvalue, t: Hashtable)
// Remove from a open hashtable. Returns the updated hash table.
//
// Values are stored in lists. The lists are stored in a zero-indexed
// vector t.v of size TABLE_SIZE.

// Map hash value to the table size
h ← x mod TABLE_SIZE

// Remove in the list at index h
l ← Array-inspect-value(t.v, h)
// Iterate over the list
p ← List-first(l)
while not List-isend(l, p) do
  if List-inspect(l, p) = x then
    // Remove the current element. The returned position is the
    // position AFTER the removed. Continue to traverse the list
    // since we may have duplicates.
    (l, p) ← List-remove(l, p)
  else
    // Advance in the list
    p ← List-next(l, p)

// Remove in list complete, now re-insert the list in the array
t.v ← Array-set-value(t.v, h, l)

return t
```


Öppen hashning (3)

- 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

Mer avancerade hashfunktioner

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

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

  - 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/Sha1sum)
  - 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*) `v` som är ett fält (*array*)
  - Varje element som är lagrat i fältet är en post med tre fält:

`hash`	nyckel-hash-värdet
`key`	nyckel-värdet
`value`	tabell-värdet

Sluten hashning, linjär teknik, insert

```
Algorithm Hash-table-closed-insert-linear(k: Key, v: Value, t: Hashtable)
// Insert the key-value pair (k, v) in the hashtable t. If the key is
// already in the table, replace the value. Returns the modified table.

// Compute the size of the hash table
size ← High(t.v) - Low(t.v) + 1
// Use the user-defined hash function to compute the key hash value
key-hash ← Key-hash-value(k)
// Map to the table size
h ← key-hash mod size

// Check each available index
for i from 0 to size - 1 do
  // Compute index with linear technique to handle collisions
  p ← (h + i) mod size
  // Inspect element at index p in the internal vector
  e ← Array-inspect-value(t.v, p)
  if e.hash = HASH_EMPTY or e.hash = HASH_REMOVED then
    // We've found a free slot; insert the value here
    r ← Create-record("hash", key-hash, "key", key, "value", v)
    t.v ← Array-set-value(t.v, p, r)
    return t
  if e.hash = key-hash then
    // We've found the hash value, now compare the actual keys
    if Key-compare(k, e.key) then
      // The keys match; replace the value associeated with the key
      r ← Create-record("hash", key-hash, "key", key, "value", v)
      t.v ← Array-set-value(t.v, p, r)
      return t

// We have checked all indices without finding an empty slot
return Error("Could not find an empty slot")
```


Sluten hashning, linjär teknik, lookup

```
Algorithm Hash-table-lookup-closed-linear(k: Key, t: Hashtable)
// Look up the key k in the hashtable t.
// If the key is found, returns (True, v), where v is the table value associated with k.
// Otherwise returns (False, None)

// Compute the size of the hash table
size ← High(t.v) - Low(t.v) + 1

// Use the user-defined hash function to compute the key hash value
key-hash ← Key-hash-value(k)
// Map to the table size
h ← key-hash mod size

// Check each available index
for i from 0 to size - 1 do
  // Compute index with linear technique to handle collisions
  p ← (h + i) mod size

  // Inspect element at index p in the internal vector
  v ← Array-inspect-value(t.v, p)
  if v.hash = HASH_EMPTY then
    // We found an empty slot; the key cannot be in the table
    return (False, None)

  if v.hash = key-hash then
    // We've found the hash value, now compare the actual keys
    if Key-compare(k, v.key) then
      // The keys match; return the value associated with the key
      return (True, v.value)

// We have checked all indices without finding a matching key
return (False, None)
```

Sluten hashning, linjär teknik, remove

```
Algorithm Hash-table-closed-remove-linear(k: Key, t: Hashtable)
// Remove the key-value pair with the key k from the hashtable t. Returns the modified table.

// Compute the size of the hash table
size ← High(t.v) - Low(t.v) + 1

// Use the user-defined hash function to compute the "object" hash value
key-hash ← Key-hash-value(k)
// Map to the table size
h ← key-hash mod size

// Check each available index
for i from 0 to size - 1 do
  // Compute index with linear technique to handle collisions
  p ← (h + i) mod size

  // Inspect element at index p in the internal vector
  e ← Array-inspect-value(t.v, p)
  if e.hash = key-hash then
    // We've found the hash value, now compare the actual keys
    if Key-compare(k, e.key) then
      // The keys match; mark the element as REMOVED
      r ← Create-record("hash", HASH_REMOVED, "key", None, "value", None)
      t.v ← Array-set-value(t.v, p, r)
      return t

return Error("Did not find the key")
```


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	Oct	May							Sep	Aug		Apr	Jul	Mar	Jun	Nov	Jan	Dec
28	31	31							31	31		30	31	31	30	30	31	31

- Kvadratisk teknik

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

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
- `Isempty` 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

Fråga

- Hur hanteras element med samma prioritet?

Formell specifikation av prioritetskö

OBS! Fel i boken!

Ax 1 Isempty (Empty)
Ax 2 ¬Isempty (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)))

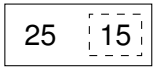
- Frågor:
  - Om R är "$<$" och två lika värden stoppas in, vilket plockas ut först?
  - Dito om R är "$\leq$".

Exempel (1)

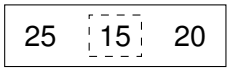
- För val=heltal, R=$<$, dvs. "*a* R *b*" är sann om $a < b$:
- `p` ← `Empty()`

- `p` ← `Insert(25,p)`

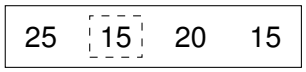
- `p` ← `Insert(15,p)`



- `p` ← `Insert(20,p)`

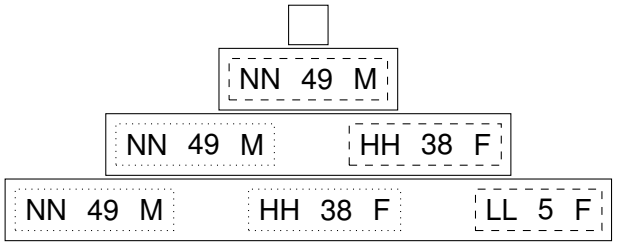


- `p` ← `Insert(15,p)`



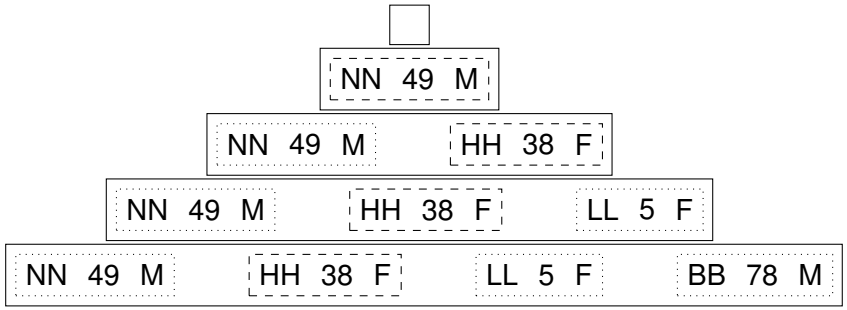
Exempel (2)

- 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

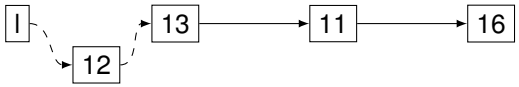
- Hur lagras elementen internt i prioritetskön?

Konstruktioner av Prioritetskö

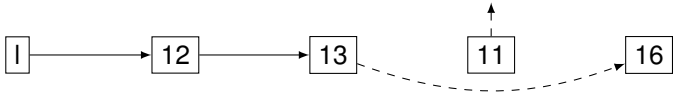
- Utgår ofta från konstruktioner av:
  - Mängd
  - Lista eller
  - Hög

Prioritetskö som osorterad lista

- val=heltal, R="$<$":
  - Insert: $O(1)$

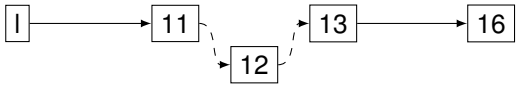


  - Inspect-First, Delete-first: $O(n)$

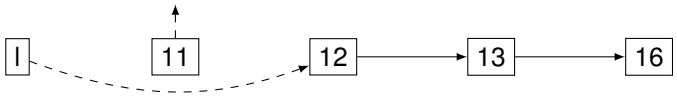


Prioritetskö som sorterad lista

- val=heltal, R="$<$":
  - Insert: $O(n)$



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

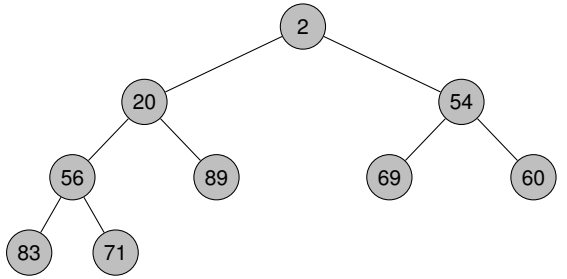


Hög (*heap*), informellt

- 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 $<$?

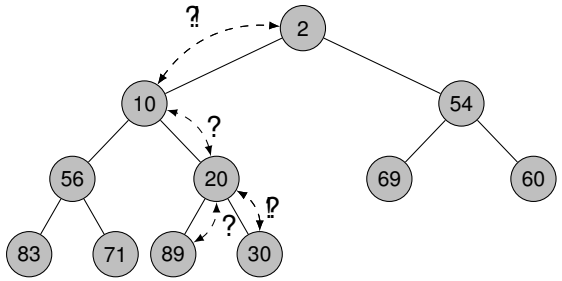


- Insättningar och borttagningar blir effektiva om dom görs så att trädet hålls komplett

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:
- Exempel: Sortera in 30:

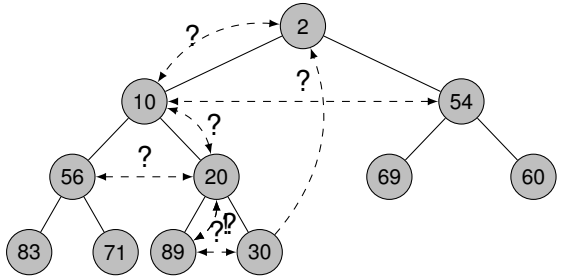


- Komplexitet för insättning av ett element i en Hög med n element?
  - $O(\log n)$

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:



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

Komplexitet för olika konstruktioner av Prioritetskö

	Insättning	Avläsning	Borttagning
Lista	$O(1)$	$O(n)$	$O(n)$
Sorterad lista	$O(n)$	$O(1)$	$O(1)$
Hög	$O(\log n)$	$O(1)$	$O(\log n)$

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
- Hjälpmedel vid traversering av graf:
  - Jfr stack och kö används vid traversering av träd