# Hashing

## IN2010 – Algoritmer og datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

# Oversikt

# Oversikt

- Vi skal lære om *hashing*

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks
  3. opprettholde en ideell størrelse på arrayet

# Oversikt

- Vi skal lære om *hashing*
- Dette er den underliggende teknikken som gir oss hashmaps og hashsets
  - Svært effektive ordbøker og mengder
- Vi ønsker å bruke *arrayer* som underliggende datastruktur
- Hashing kan brytes opp i tre problemer
  1. gjøre en vilkårlig verdi om til et tall som brukes som en indeks
  2. hvordan håndtere to verdier som får samme indeks
  3. opprettholde en ideell størrelse på arrayet
- Målet er å få effektiv innsetting, oppslag og sletting

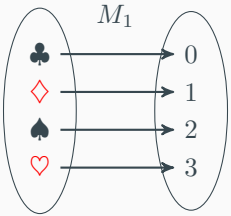
# Ordbok

# Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi

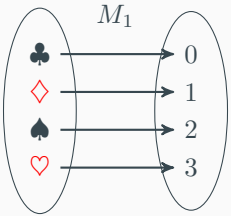
# Ordbok

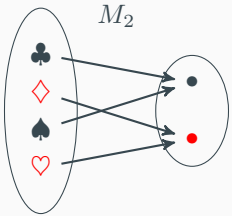
- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi



# Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi



# Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi




- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok $M$

# Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi




- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok $M$
  - $M[k] \leftarrow v$ eller $\texttt{put}(M, k, v)$ Assosiere nøkkelen $k$ med verdien $v$

# Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi




- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok $M$
  - $M[k] \leftarrow v$ eller `put`$(M, k, v)$ — Assosiere nøkkelen $k$ med verdien $v$
  - $M[k]$ eller `get`$(M, k)$ — Hente ut verdien som nøkkelen $k$ er assosiert med

# Ordbok

- En *ordbok* (eller et *map*), helt generelt, assosierer en nøkkel med nøyaktig én verdi




- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok $M$
  - $M[k] \leftarrow v$ eller $\texttt{put}(M, k, v)$ — Assosiere nøkkelen $k$ med verdien $v$
  - $M[k]$ eller $\texttt{get}(M, k)$ — Hente ut verdien som nøkkelen $k$ er assosiert med
  - $M \setminus k$ eller $\texttt{Remove}(M, k)$ — Fjerne assosiasjonen forbundet med nøkkelen $k$

# Hashmap

- Et hashmap er en måte å implementere en ordbok

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på $A$ er

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på $A$ er
- Derfor vil vi få *kollisjoner*

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på $A$ er
- Derfor vil vi få *kollisjoner*
  - Altså at to ulike nøkler blir konvertert til det samme tallet $i$

# Hashmap

- Et hashmap er en måte å implementere en ordbok
- Vi bruker et array $A$, sammen med en *hashfunksjon* $h$
- Hashfunksjonen konverterer en nøkkel $k$ til et tall $i$ der $0 \leq i < |A|$
  - Vi kaller denne konverteringen for «å hashe»
- Som regel finnes det utrolig mange *mulige* nøkler
  - Det finnes for eksempel uendelig mange forskjellige strenger
- Det er umulig å koke uendelig mange ting ned til $|A|$ tall
  - uansett hva størrelsen på $A$ er
- Derfor vil vi få *kollisjoner*
  - Altså at to ulike nøkler blir konvertert til det samme tallet $i$
- Vi skal se på to måter å håndtere kollisjoner på

# Hashfunksjoner

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input
  - og returnerer et heltall slik at $0 \leq h(k, N) < N$

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input
  - og returnerer et heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input
  - og returnerer et heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input
  - og returnerer et heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den bør gi få *kollisjoner* (altså være godt distribuert)

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input
  - og returnerer et heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den bør gi få *kollisjoner* (altså være godt distribuert)
  - Ulike input bør hashe til ulike output så ofte som mulig

# Hashfunksjoner

- En *funksjon* returner samme output for et gitt input *hver gang*
  - Altså er mange prosedyrer og metoder *ikke funksjoner*
- En *hashfunksjon* $h$
  - får en nøkkel $k$ og et positivt heltall $N$ som input
  - og returnerer et heltall slik at $0 \leq h(k, N) < N$
- Den må være *konsistent* (altså være en *funksjon*)
  - Samme input gir *alltid* samme output
- Den bør gi få *kollisjoner* (altså være godt distribuert)
  - Ulike input bør hashe til ulike output så ofte som mulig
- Ulike datatyper krever ulike hashfunksjoner

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm:** En inkonsistent hashfunksjon

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2     return Random(0, N − 1)
```

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm:** En inkonsistent hashfunksjon

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm:** En inkonsistent hashfunksjon

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm: En inkonsistent hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

**Algorithm: En dårlig distribuert hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm: En inkonsistent hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

**Algorithm: En dårlig distribuert hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

- Denne hashfunksjonen er konsistent, men alt kolliderer med alt!

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm: En inkonsistent hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

**Algorithm: En dårlig distribuert hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

- Denne hashfunksjonen er konsistent, men alt kolliderer med alt!

# Hashfunksjoner – Eksempler på dårlige hashfunksjoner

**Algorithm: En inkonsistent hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Inconsistent(k, N)
2 |   return Random(0, N − 1)
```

- Tilfeldige verdier er ideelt for å unngå kollisjoner, men dette er ikke en funksjon!

**Algorithm: En dårlig distribuert hashfunksjon**

**Input:** En nøkkel $k$ og et positivt heltall $N$
**Output:** Et heltall $i$ slik at $0 \leq i < N$

```
1 Procedure Collider(k, N)
2 |   return 0
```

- Denne hashfunksjonen er konsistent, men alt kolliderer med alt!
- Vi ønsker funksjoner som ikke lider av noen av disse problemene

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$
1 **Procedure** `HashStringBad`$(s, N)$

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
```

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
```

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo* $N$

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm:** En litt dårlig hashfunksjon på strenger

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo* $N$
- Denne er konsistent og kan distribuere greit

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm:** En litt dårlig hashfunksjon på strenger

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo* $N$
- Denne er konsistent og kan distribuere greit
  - Hvorfor er den allikevel ikke god i praksis?

# Distribusjon: strenger

- For å hashe en streng kan vi se på hver bokstav som et tall

**Algorithm: En litt dårlig hashfunksjon på strenger**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashStringBad(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← h + charToInt(c)
5     return h mod N
```

- Her summerer vi alle tallverdiene for bokstavene
- Til slutt returnerer vi denne summen *modulo* $N$
- Denne er konsistent og kan distribuere greit
  - Hvorfor er den allikevel ikke god i praksis?
  - Hint: $a + b = b + a$

# Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

# Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

# Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

**ALGORITHM: EN GOD HASHFUNKSJON PÅ STRENGER**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
```

# Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

**ALGORITHM: EN GOD HASHFUNKSJON PÅ STRENGER**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     return h mod N
```

# Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

**ALGORITHM: EN GOD HASHFUNKSJON PÅ STRENGER**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     return h mod N
```

- Denne er konsistent og distribuerer ganske godt!

# Distribusjon: strenger

- Vi fortsetter med samme idé, men introduserer litt mer kaos!

**ALGORITHM: EN GOD HASHFUNKSJON PÅ STRENGER**

**Input:** En streng $s$ og et positivt heltall $N$
**Output:** Et heltall $h$ slik at $0 \leq h < N$

```
1 Procedure HashString(s, N)
2     h ← 0
3     for every letter c in s do
4         h ← 31 · h + charToInt(c)
5     return h mod N
```

- Denne er konsistent og distribuerer ganske godt!
  - Det er hashfunksjonen som brukes i Java

# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok

# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)

# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder 235886 ord

# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder $235886$ ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av $N$

# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder $235886$ ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av $N$
- Hvis vi lar $N = 235886$ så får vi at

# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder $235886$ ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av $N$
- Hvis vi lar $N = 235886$ så får vi at
  - `HashStringBad("algorithm", N)` hasher til $967$, og kolliderer med $577$ andre ord

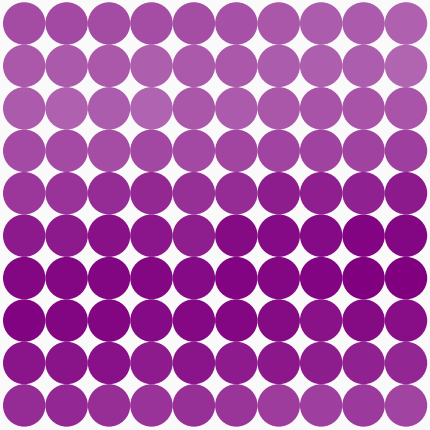
# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder $235886$ ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av $N$
- Hvis vi lar $N = 235886$ så får vi at
  - `HashStringBad("algorithm", N)` hasher til $967$, og kolliderer med $577$ andre ord
  - `HashString("algorithm", N)` hasher til $184369$, og kolliderer med ett annet ord

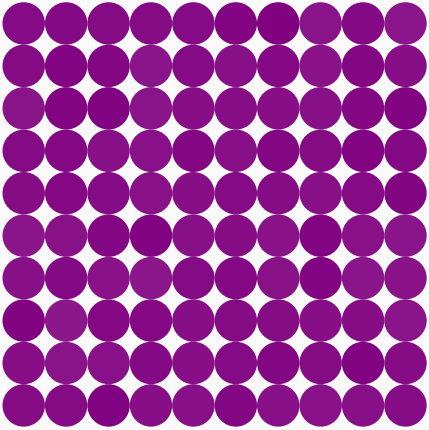
# Distribusjon: `HashStringBad` vs `HashString`

- La oss hashe alle ordene i en ordbok
  - (den som ligger her på mange maskiner: `/usr/share/dict/words`)
- Den inneholder $235886$ ord
- Vi kan teste hvor mange kollisjoner vi får for ulike verdier av $N$
- Hvis vi lar $N = 235886$ så får vi at
  - `HashStringBad("algorithm", N)` hasher til $967$, og kolliderer med $577$ andre ord
  - `HashString("algorithm", N)` hasher til $184369$, og kolliderer med ett annet ord
- Vi skal se illustrasjoner som viser resultatene av å bruke hver hashfunksjon på alle ordene i ordboken for ulike verdier av $N$

# Distribusjon: `HashStringBad` vs `HashString` (N=100)



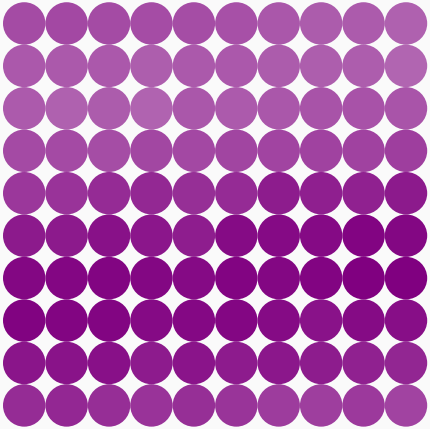
`HashStringBad`



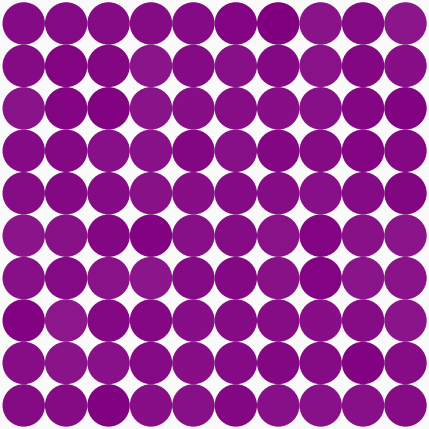
`HashString`

- Hver prikk svarer til tallene $0, 1, \ldots, N-1$

# Distribusjon: `HashStringBad` vs `HashString` (N=100)



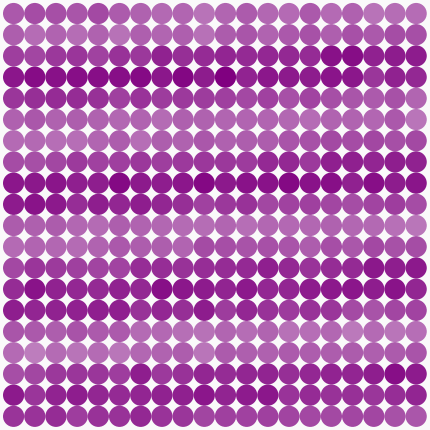
`HashStringBad`



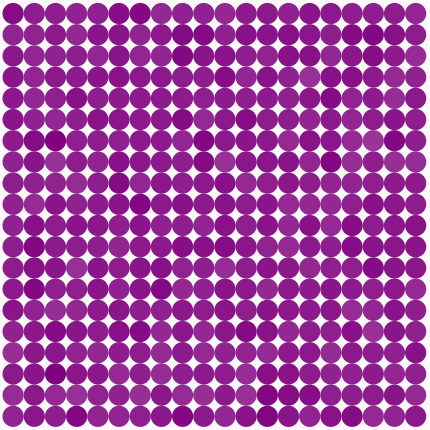
`HashString`

- Hver prikk svarer til tallene $0, 1, \ldots, N-1$
- Når et ord hasher til et tall $i$, så blir fargen til den $i$-te prikken litt sterkere

# Distribusjon: HashStringBad vs HashString (N=400)

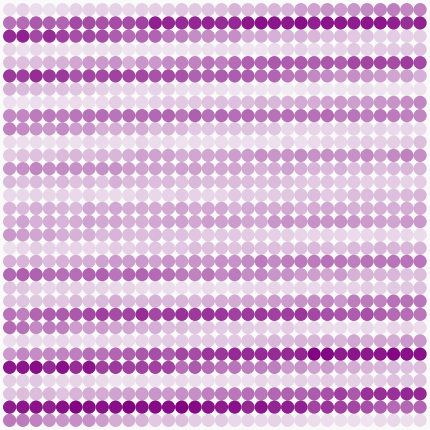


HashStringBad

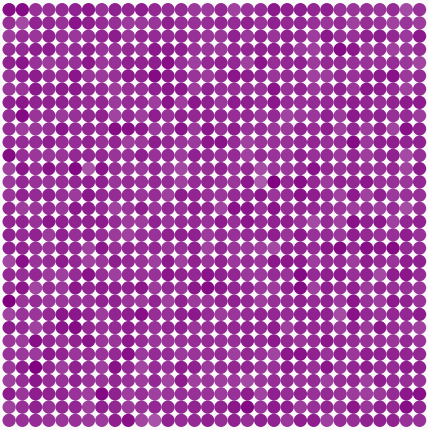


HashString

# Distribusjon: HashStringBad vs HashString (N=1024)

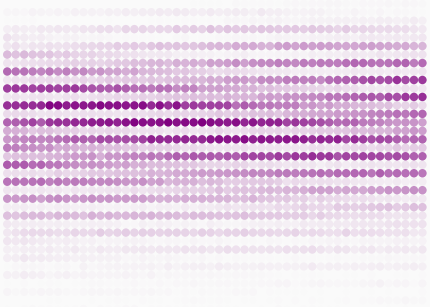


HashStringBad

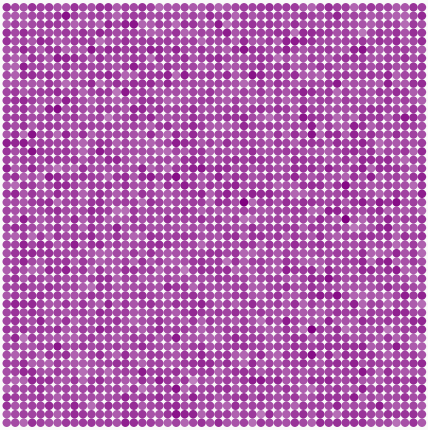


HashString

# Distribusjon: HashStringBad vs HashString (N=2500)

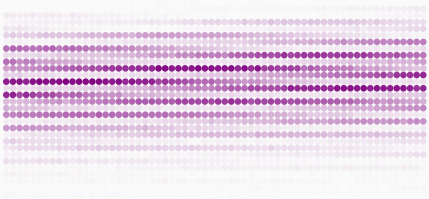


HashStringBad

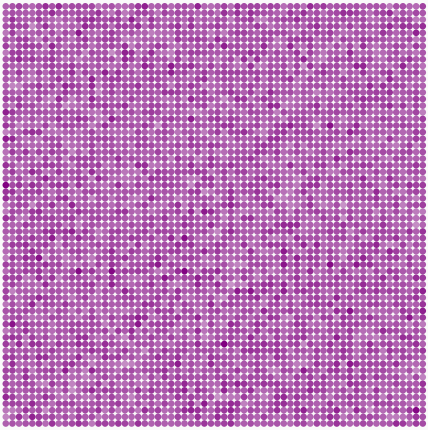


HashString

# Distribusjon: `HashStringBad` vs `HashString` (N=4096)



`HashStringBad`



`HashString`

# Kollisjonshåndtering

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om
- I denne seksjonen antar vi at:

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om
- I denne seksjonen antar vi at:
  - $A$ er et array med størrelse $N$

# Kollisjonshåndtering

- Ved «Separate chaining» lar vi hver plass i arrayet peke til en «bøtte»
  - Vi tar utgangspunkt i at hver bøtte er en *lenket liste*
  - (Man kan for eksempel heller bruke binære søketrær)
- Ved «Linear probing» bruker vi kun arrayet
  - Ved kollisjoner ser vi etter neste ledige plass i arrayet
  - Dette er enkelt, men ved sletting må vi tenke oss litt om
- I denne seksjonen antar vi at:
  - $A$ er et array med størrelse $N$
  - $A$ inneholder mindre enn $N$ elementer

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner `null`

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen $k$ i $B$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen $k$ i $B$
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen $k$ i $B$
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$

# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen $k$ i $B$
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner

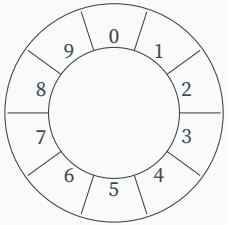
# Kollisjonshåndtering – Separate chaining

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, opprett en liste og sett inn $(k, v)$
  3. Ellers settes $(k, v)$ inn på slutten av $B$
     - Hvis det finnes en node med nøkkel $k$ på veien, erstattes verdien med $v$
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner `null`
  3. Hvis ikke, slå opp på nøkkelen $k$ i $B$
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. La $B \leftarrow A[i]$
  2. Hvis $B$ er `null`, returner
  3. Hvis ikke, slett noden med nøkkelen $k$ i $B$

# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

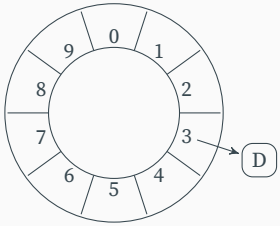
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R | |
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 | |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

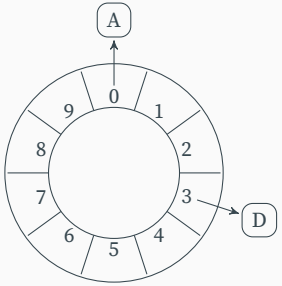
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

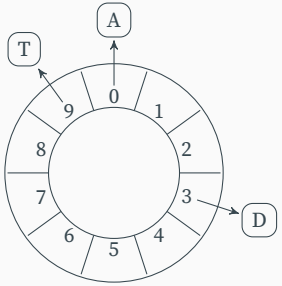
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

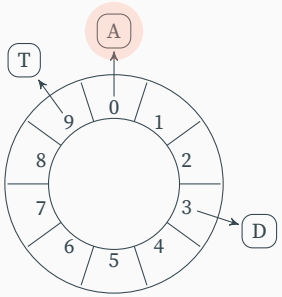
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

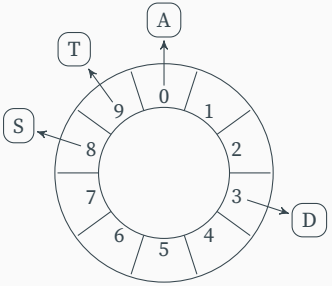
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

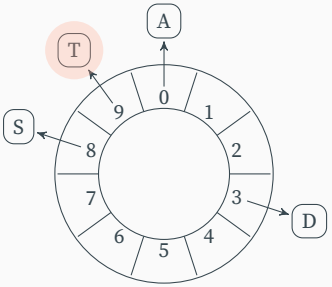
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

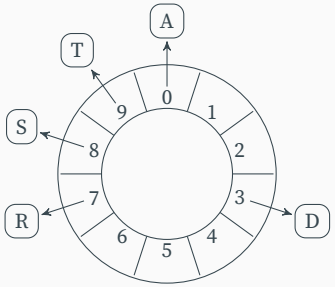
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

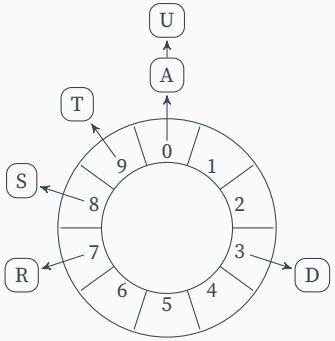
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

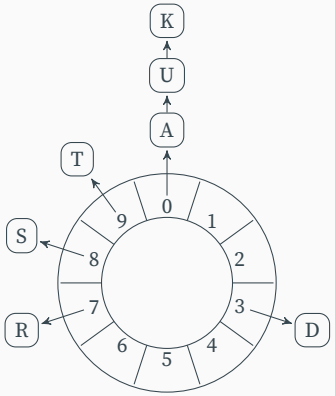
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

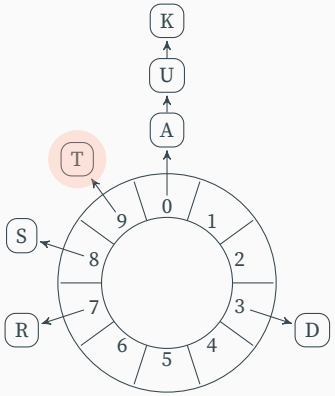
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

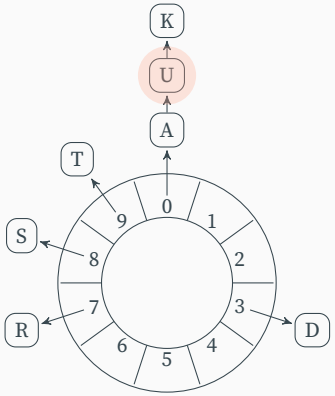
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

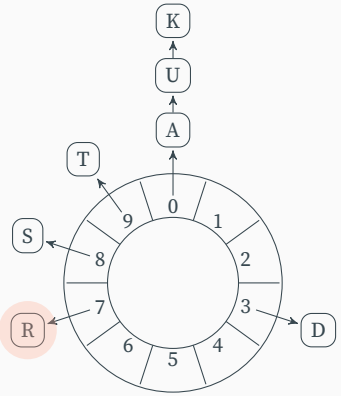
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

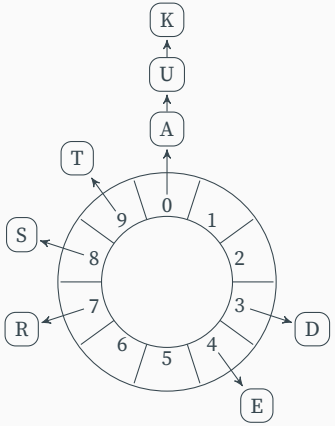
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

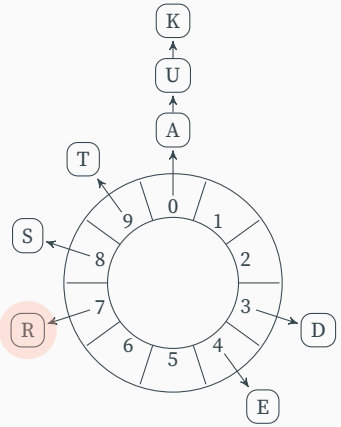
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

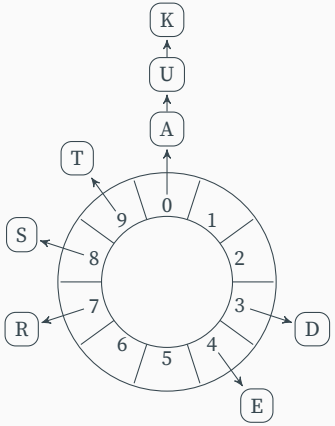
| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Separate chaining (eksempel)

| $k$ | $h(k, 10)$ |
|---|---|
| A | 0 |
| B | 1 |
| C | 2 |
| D | 3 |
| E | 4 |
| F | 5 |
| G | 6 |
| H | 7 |
| I | 8 |
| J | 9 |
| K | 0 |
| L | 1 |
| M | 2 |
| N | 3 |
| O | 4 |
| P | 5 |
| Q | 6 |
| R | 7 |
| S | 8 |
| T | 9 |
| U | 0 |
| V | 1 |
| W | 2 |
| X | 3 |
| Y | 4 |
| Z | 5 |

| $k$: | D | A | T | A | S | T | R | U | K | T | U | R | E | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h(k, 10)$: | 3 | 0 | 9 | 0 | 8 | 9 | 7 | 0 | 0 | 9 | 0 | 7 | 4 | 7 |



# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen på $A[i]$ er ulik $k$, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen på $A[i]$ er ulik $k$, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)
  3. Hvis nøkkelen på $A[i]$ er lik $k$, sett $A[i]$ til `null`

# Kollisjonshåndtering – Linear probing

- **Innsetting:** gitt en nøkkel $k$ som hasher til $i$, og en verdi $v$
  1. Hvis $A[i]$ er `null`, sett inn $(k, v)$ på plass $i$ og returner
  2. Hvis nøkkelen til $A[i]$ er $k$, sett inn $(k, v)$ på plass $i$ og returner
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Oppslag:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen til $A[i]$ er $k$, returner verdien på $A[i]$
  3. Gå til neste plass ($i \leftarrow i + 1 \bmod N$) og gå til steg 1
- **Sletting:** gitt en nøkkel $k$ som hasher til $i$
  1. Hvis $A[i]$ er `null`, returner `null`
  2. Hvis nøkkelen på $A[i]$ er ulik $k$, gå til steg 1 med ($i \leftarrow i + 1 \bmod N$)
  3. Hvis nøkkelen på $A[i]$ er lik $k$, sett $A[i]$ til `null`
  4. *Tett hullet*

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Merkene forsvinner ved neste rehash (se neste seksjon)

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Merkene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass $i$, tett hullet (uten juks)

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Merkene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass $i$, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til $i$ eller tidligere

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Merkene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass $i$, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til $i$ eller tidligere
     - Hvis treffer en `null`, kan vi avslutte

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Merkene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass $i$, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til $i$ eller tidligere
     - Hvis treffer en `null`, kan vi avslutte
     - Finner vi en slik nøkkel flyttes den (sammen med verdien) til plass $i$

# Kollisjonshåndtering – Linear probing (tett hullet)

- Etter fjerning må vi passe på at vi tetter eventuelle hull
- Husk at alle algoritmene for linear probing terminerer ved tomme plasser
  - Derfor kan vi miste elementer hvis vi ikke tetter hullet
- Vi har to strategier:
  1. Markér plassen som slettet
     - Ved søk vil vi ikke stoppe på markerte felter
     - Ved innsetting vil vi anse markerte felter som ledig
     - Merkene forsvinner ved neste rehash (se neste seksjon)
  2. Hvis hullet er på plass $i$, tett hullet (uten juks)
     - Søk etter en nøkkel som hasher til $i$ eller tidligere
     - Hvis treffer en `null`, kan vi avslutte
     - Finner vi en slik nøkkel flyttes den (sammen med verdien) til plass $i$
     - Så må vi tette det nye hullet på samme måte

# Effektivitet

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer $n$ i hashmappen og størrelsen på arrayet $N$

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer $n$ i hashmappen og størrelsen på arrayet $N$
  - altså er load factor gitt ved $\frac{n}{N}$

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer $n$ i hashmappen og størrelsen på arrayet $N$
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer $n$ i hashmappen og størrelsen på arrayet $N$
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt
  - men antageligvis ligger den mellom $\frac{1}{2}$ og $\frac{3}{4}$

# Effektivitet – Load factor

- For at et hashmap skal være effektivt må vi velge en god størrelse på arrayet
- Dersom arrayet er lite (i forhold til antall elementer) vil vi kunne
  - for **separate chaining** få lange lister, som bruker lineær tid på alle operasjoner
  - for **linear probing** få lange segmenter uten hull, som igjen gir lineær tid
- Dersom arrayet er for stort, sløser vi med plass
- Load factor angir forholdet mellom antall elementer $n$ i hashmappen og størrelsen på arrayet $N$
  - altså er load factor gitt ved $\frac{n}{N}$
- Å finne en ideell load factor bør avgjøres eksperimentelt
  - men antageligvis ligger den mellom $\frac{1}{2}$ og $\frac{3}{4}$
  - altså bør arrayet holdes litt mer en halvfullt

# Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing

# Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:

# Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
    1. lage et større array

# Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet

# Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet
- Det samme kan man gjøre dersom arrayet blir for tomt

# Effektivitet – Rehashing

- Dersom arrayet blir for fullt (altså for høy load factor) gjør vi rehashing
- Det betyr å:
  1. lage et større array
  2. sette inn alle elementene fra det forrige arrayet
- Det samme kan man gjøre dersom arrayet blir for tomt
  - Ofte gjør man ikke det i praksis

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps
- For hashmaps bruker vi noe som kalles *forventet amortisert* kjøretidsanalyse

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps
- For hashmaps bruker vi noe som kalles *forventet amortisert* kjøretidsanalyse
  - Å gjøre forventet amortisert kjøretidsanalyse er ikke pensum i IN2010

# En uformell kjøretidsanalyse

- Ved å gjøre en ordinær kjøretidsanalyse vil vi se på hvor mange steg algoritmene vil bruke i *verste tilfelle*
- For hashmap gir en slik analyse lineær tid på alle operasjoner
  - Det holder å anta at alle nøkler hasher til samme tall
  - Separate chaining oppfører seg da som lenkede lister
  - Linear probing oppfører seg som uordnede arrayer
- Alikevel påstår vi at hashmaps er *svært effektive*
  - Dette er en indikasjon på at en verste tilfelle $\mathcal{O}$-analyse ikke gir et godt bilde av effektiviteten av hashmaps
- For hashmaps bruker vi noe som kalles *forventet amortisert* kjøretidsanalyse
  - Å gjøre forventet amortisert kjøretidsanalyse er ikke pensum i IN2010
  - Men vi skal vite hva det betyr i konteksten av hashing

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - Dette tallet blir forsvinnende lite når $m$ vokser

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - Dette tallet blir forsvinnende lite når $m$ vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - Dette tallet blir forsvinnende lite når $m$ vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - Ethvert som arrayet fylles opp, så øker sjansen for kollisjoner

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - Dette tallet blir forsvinnende lite når $m$ vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - Ethvert som arrayet fylles opp, så øker sjansen for kollisjoner
  - Sjansen for at kollisjonene oppstår på de samme posisjonene er fremdeles liten

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - Dette tallet blir forsvinnende lite når $m$ vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - Ethvert som arrayet fylles opp, så øker sjansen for kollisjoner
  - Sjansen for at kollisjonene oppstår på de samme posisjonene er fremdeles liten
- Oppslag og sletting er *forventet* $\mathcal{O}(1)$

# En uformell (forventet) kjøretidsanalyse

- La oss anta at vi har en god hashfunksjon som gir en uniformt tilfeldig fordeling
- Vi ønsker å vise at sjansen er forsvinnende liten for at *mange* elementer hasher til samme posisjon
- Sjansen for at en nøkkel $k$ hasher til $i$ er $\frac{1}{N}$
- Sjansen for at vi har mange nøkler $k_1, k_2, \ldots, k_m$ som alle hasher til $i$ blir

$$\overbrace{\frac{1}{N} \cdot \frac{1}{N} \cdots \frac{1}{N}}^{m} = \left(\frac{1}{N}\right)^m$$

  - Dette tallet blir forsvinnende lite når $m$ vokser
- Generelt er sjansen for kollisjoner liten så lenge det er god plass i arrayet
  - Ethvert som arrayet fylles opp, så øker sjansen for kollisjoner
  - Sjansen for at kollisjonene oppstår på de samme posisjonene er fremdeles liten
- Oppslag og sletting er *forventet* $\mathcal{O}(1)$
  - Innsetting er *forventet amortisert* $\mathcal{O}(1)$

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der $c$ er en konstant

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der $c$ er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der $c$ er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde $c$ ekstra steg

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der $c$ er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde $c$ ekstra steg
  - Hver innsetting bruker fremdeles konstant tid, siden $\mathcal{O}(1 + c) = \mathcal{O}(1)$

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der $c$ er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde $c$ ekstra steg
  - Hver innsetting bruker fremdeles konstant tid, siden $\mathcal{O}(1 + c) = \mathcal{O}(1)$
- Merk at innsetting i hashmaps vil være inneffektiv en sjelden gang

# En uformell (amortisert) kjøretidsanalyse

- Når arrayet blir for fullt gjør vi en rehash
  - Altså, konstruerer et nytt array og setter inn alle elementene på nytt
  - Dette er definitivt lineær tid!
- Innsettingen som forårsaker en rehash er lineær tid
  - Dette kan ses på som verste tilfelle
- I amortisert kjøretidsanalyse ser vi heller på *alle innsettingene som ledet opp til en rehash* under ett
- Anta at en rehash skjer etter $n$ innsettinger
  - der hver innsetting frem til rehashen bruker $\mathcal{O}(1)$ steg
- Anta videre at en rehash bruker $c \cdot n$ steg, der $c$ er en konstant
- Vi kan nå *fordele* kostnaden av rehashen på *alle innsettingene*
  - Altså kan vi si at hver innsetting gjorde $c$ ekstra steg
  - Hver innsetting bruker fremdeles konstant tid, siden $\mathcal{O}(1 + c) = \mathcal{O}(1)$
- Merk at innsetting i hashmaps vil være inneffektiv en sjelden gang
  - Dette kan være en viktig å være klar over for sanntidsapplikasjoner

# Effektivitet

- Hashmaps er utrolig raske i praksis, og ekstremt anvendelige

# Effektivitet

- Hashmaps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $\mathcal{O}(n)$ på alle operasjoner

# Effektivitet

- Hashmaps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $\mathcal{O}(n)$ på alle operasjoner
- Men dersom man antar en god hashfunksjon får vi *forventet amortisert* $\mathcal{O}(1)$

# Effektivitet

- Hashmaps er utrolig raske i praksis, og ekstremt anvendelige
- I en verste tilfelle analyse har de $\mathcal{O}(n)$ på alle operasjoner
- Men dersom man antar en god hashfunksjon får vi *forventet amortisert* $\mathcal{O}(1)$
- Rehashing tar $\mathcal{O}(n)$ tid, men det skjer sjeldent