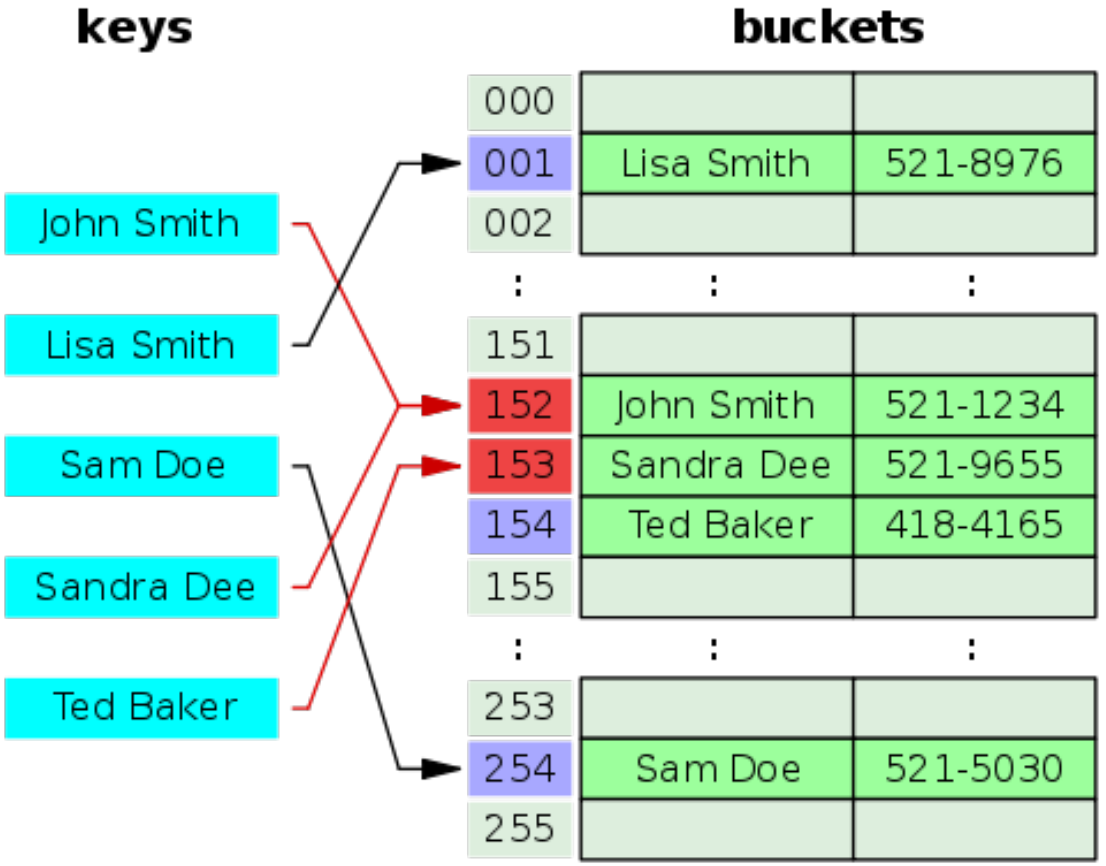
Hashing:
Håndtering av kollisjoner



Innsetting av dataelement i hashtabell

- Algoritme:

  1. Bruk en hashfunksjon til å beregne hashverdi basert på dataelementets nøkkelverdi
  2. Sett inn dataelementet i hashtabellen

- Innsetting i hashtabell er $O(1)$ hvis problemet med kollisjoner kan løses med en $O(1)$ operasjon
- Innsetting blir langsommere enn $O(1)$ når hashtabellen blir «for full» av data og det blir mange kollisjoner

Load factor

- Hashing er svært effektivt så lenge hashtabellen har «mange ledige plasser» og det er lite kollisjoner
- «Load factor» *L* er et mål på hvor full en hashtabell er

  $L = n / h$

  n: Antall elementer lagret i hashtabellen

  h: hashlengden

- Tolking av load factor:

  $L = 0.5$ Halvfull hashtabell

  $L < 1.0$ Færre datalelementer enn arrayplasser

  $L > 1.0$ Flere datalelementer enn arrayplasser

To metoder for håndtering av kollisjoner

- Åpen adressering * :
  - Hvis en indeks i hash-tabellen er opptatt, legges elementet et annet sted i tabellen på en eller annen systematisk måte (f.eks. neste ledige)
  - Load factor maks 1.0
- Kjeding ** :
  - Hvert element i hash-tabellen er en lenket liste (søketre?) som inneholder alle elementer med samme hashverdi
  - Load factor kan være større enn 1.0

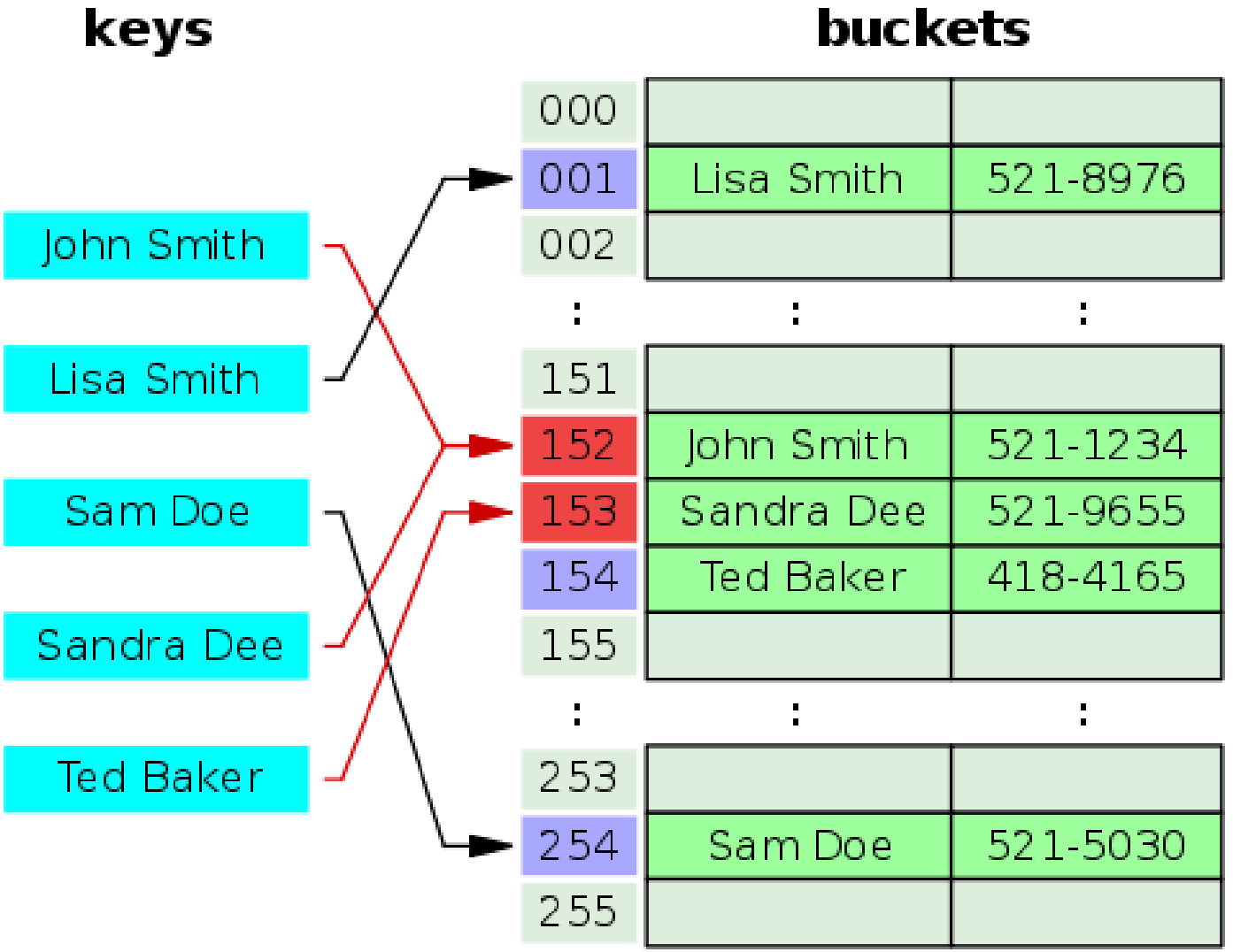
*: aka. "open addressing", "closed hash table"
**: aka. "chaining", "closed addressing", "open hash table"

Åpen adressering

- Hvis hashindeksen som beregnes er opptatt, gjør vi en «probing» (et søk) etter en *annen* ledig indeks i hashtabellen der vi kan legge elementet
- Probingen må være systematisk/deterministisk, slik at vi kan *finne igjen* elementet ved søking
- Må kunne håndtere muligheten for at vi *ikke* finner noen ledig plass i tabellen på en robust måte

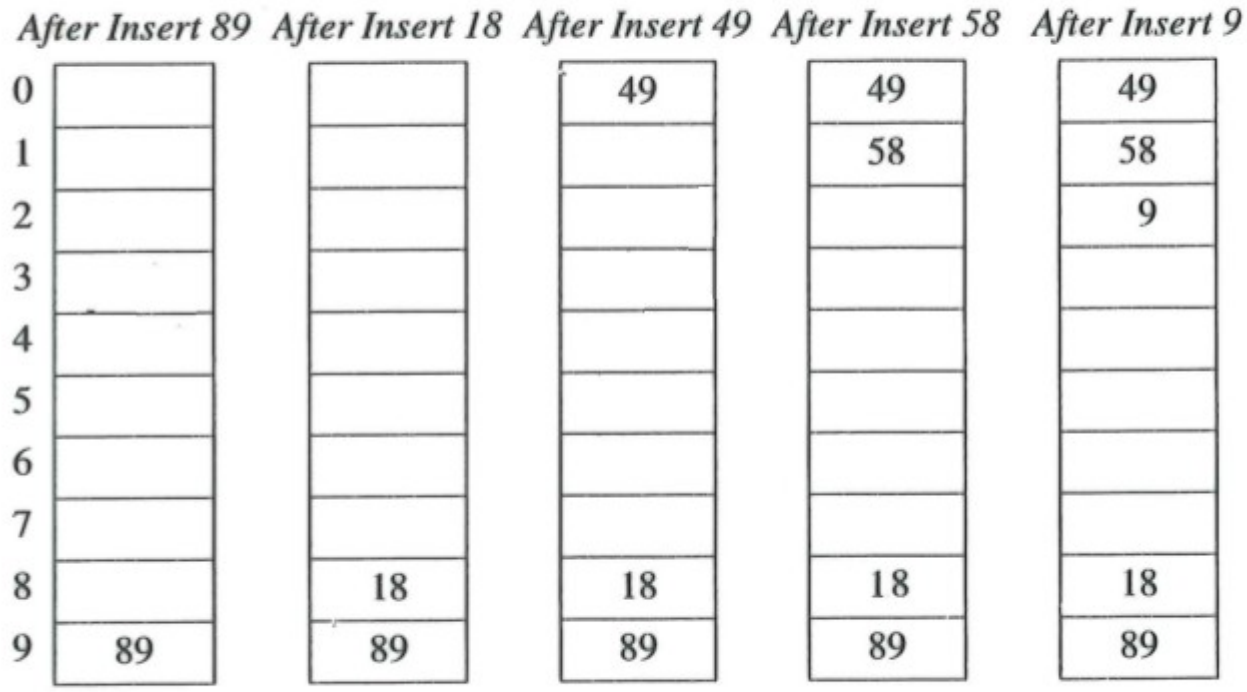
Enkleste variant av åpen adressering: Lineær probing

- Beregn dataelementets hashverdi *h*
- Hvis indeks *h* i hashtabellen er opptatt:
  - Sett inn nytt dataelement på *første* ledige indeks *etter* indeks *h*
  - Hvis indeks *h* + 1 er opptatt, prøv å sette inn på indeks *h* + 2, *h* + 3, *h* + 4, osv., inntil en ledig plass er funnet
  - Gjør en «wrap-around» (fortsett med indeks 0) hvis vi kommer til slutten av tabellen


keys
buckets
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
155
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

Lineær probing: Eksempel

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```



Linear probing hash table after each insertion

Animasjon av åpen adressering med lineær probing

- Eksempel med hashlengden lik 29
- Kan velge mellom hashing av heltall eller strenger
- Enkel hashfunksjon for heltall:

```
hash(key) = key % hash_lengde
```

- Demo: Closed Hashing

Enkel implementasjon av lineær probing

- Hashing av tekststrenger: `hashLinear.java`
- Implementerer innsetting og søking i hashtabellen
- Fjerning av data er ikke implementert
- Ingen håndtering av full hashtabell, bare "gir opp"
- Koden for *innsetting* skal skrives om i obligatorisk oppgave 5, til å implementere to ulike varianter av lineær probing:
  - "Last come, first served"-hashing
  - "Robin Hood"-hashing

Fordeler med lineær probing

- Enkelt å programmere
- Svært rask beregning av probes (kun én addisjon)
- Meget effektiv så lenge det er god plass i hashtabellen
- "Robin Hood"-varianten av lineær probing er oftest effektiv også for høye verdier av load factor

Ulemper ved lineær probing

- Lite effektiv håndtering av clustering / klumping
- Sprer ikke data som hashes til samme område i hashtabellen – «primary clustering»
- Alle elementer som har samme hashverdi vil bli liggende i en «klump» i tabellen

Effektivitet av lineær probing

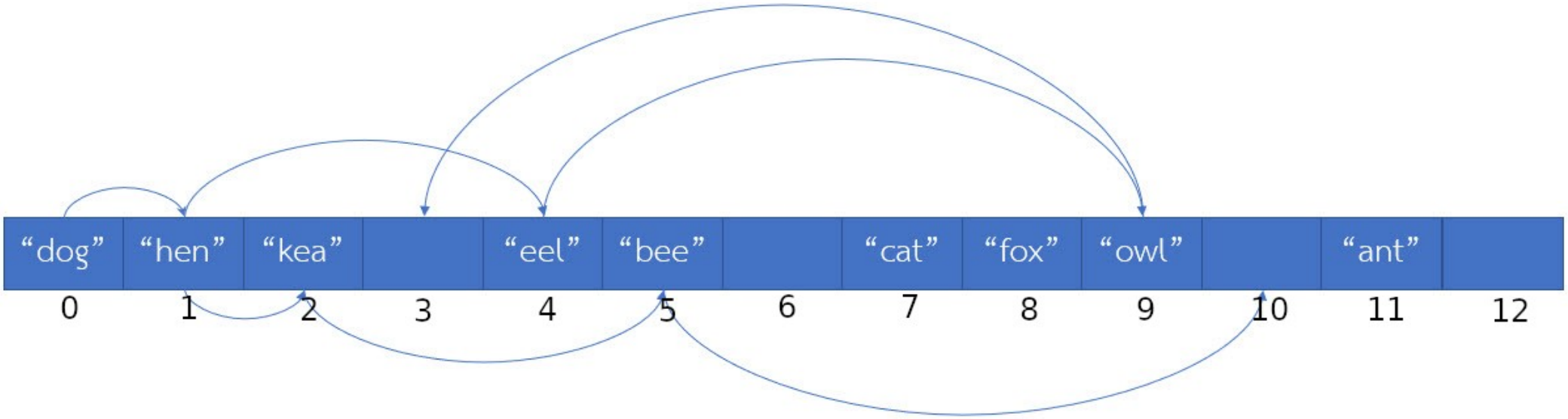
- Det kan bevises at:
  - Hvis dataene som settes inn er «rimelig tilfeldige» og load factor er ≤ 0.5 (maks. halvfull hashtabell), vil lineær probing *alltid* gi hashtabeller med $O(1)$ effektivitet
- Innsetting og søking er lite effektivt når tabellen begynner å bli full:
  - Lineær probing får problemer for load factor > 0.7
  - Får lange «opptatte sekvenser» i tabellen som må gås gjennom for å finne en ledig plass

Åpen adressering med kvadratisk probing

- Beregn dataelementets hashverdi *h*
- Hvis indeks *h* i hashtabellen er opptatt:
  - Forsøk å sette inn på indeks *h* + 1
  - Hvis indeks *h* + 1 er opptatt, prøv å sette inn på indeksene:

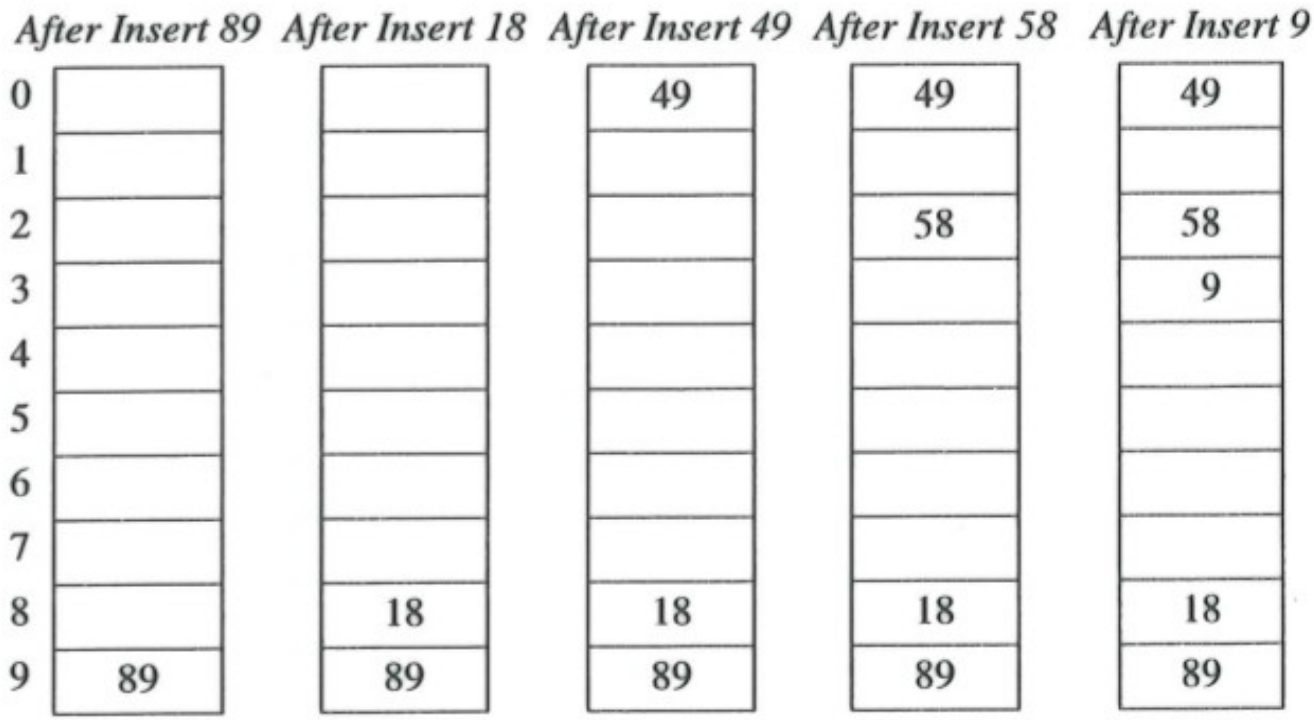
    h + 4, *h* + 9, *h* + 16, *h* + 25, *h* + 36, osv.,

    inntil en ledig plass finnes
  - Gjør en «wrap-around» (fortsett med indeks 0) hvis vi havner utenfor hashtabellen


“dog”
“hen”
“kea”
“eel”
“bee”
“cat”
“fox”
“owl”
“ant”
0
1
2
3
4
5
6
7
8
9
10
11
12

Kvadratisk probing: Eksempel

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```



Quadratic probing hash table after each insertion (note that the table size is poorly chosen because it is not a prime number)

Animasjon av kvadratisk probing

- Eksempel med hashlengden lik 29
- Kan velge mellom hashing av heltall eller strenger
- Enkel hashfunksjon for heltall:

```
hash(key) = key % hash_lengde
```

- Demo: Closed Hashing

Enkel implementasjon av kvadratisk probing

- Hashing av tekststrenger: `hashQuadratic.java`
- Implementerer innsetting og søking i hashtabellen
- Fjerning av data er ikke implementert
- Ingen håndtering av full hashtabell, bare "gir opp"

Kvadratisk probing: Fordeler og ulemper

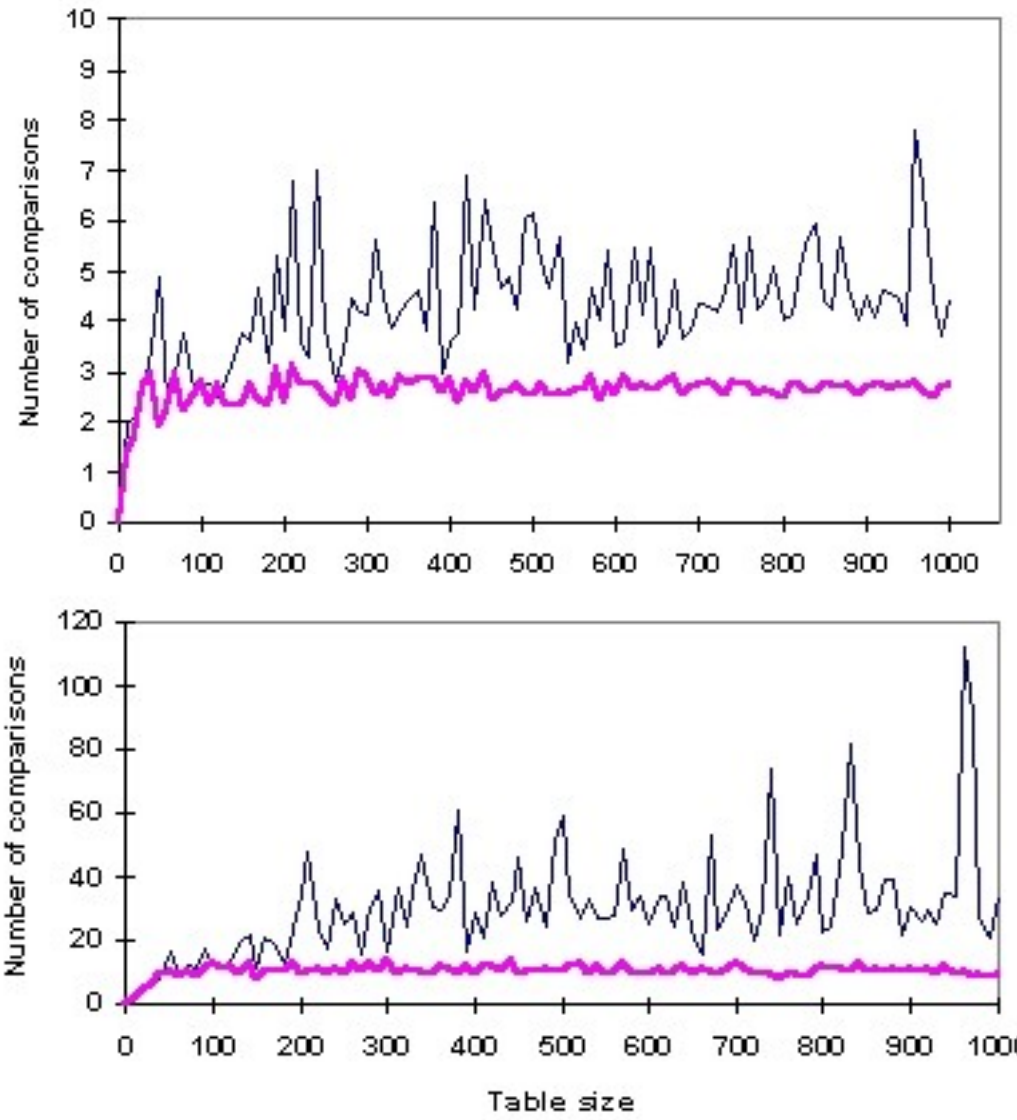
- Fordeler:
  - Sprer elementene bedre enn lineær probing
  - Løser opp «primary clusters» ved å flytte elementer med lik hashverdi langt fra hverandre
- Ulemper:
  - Beregning av kvadratiske «probes» er mer kostbart enn lineære
  - Løser ikke opp «secondary clusters»: Elementer som har hashverdier som ligger *nære* hverandre i hashtabellen, vil i liten grad spres

Effektivitet av kvadratisk probing

- Det kan bevises at:
  - Hvis load factor er ≤ 0.5 og hashlengden er et primtall, vil kvadratisk probing *alltid* gi hashtabeller med $O(1)$ effektivitet
- Erfaring har vist at effektivitetet er svært god for load factor opp til minst 0.8
- Matematisk analyse av kvadratisk probing er vanskeligere enn for lineær probing og ikke komplett
- De mest brukte løsningene er derfor i stor grad basert på «best practice»

Lineær og kvadratisk probing: Sammenligning av effektivitet av søking

Konstant load factor lik 0.9, varierende hashlengder

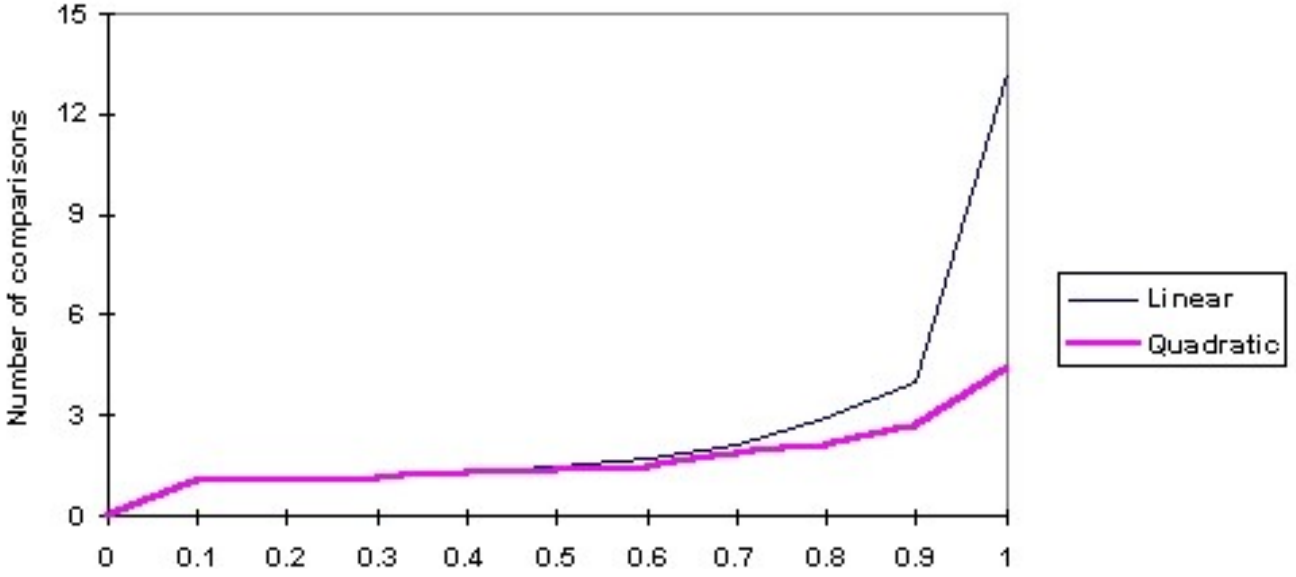


Søk der element ble funnet

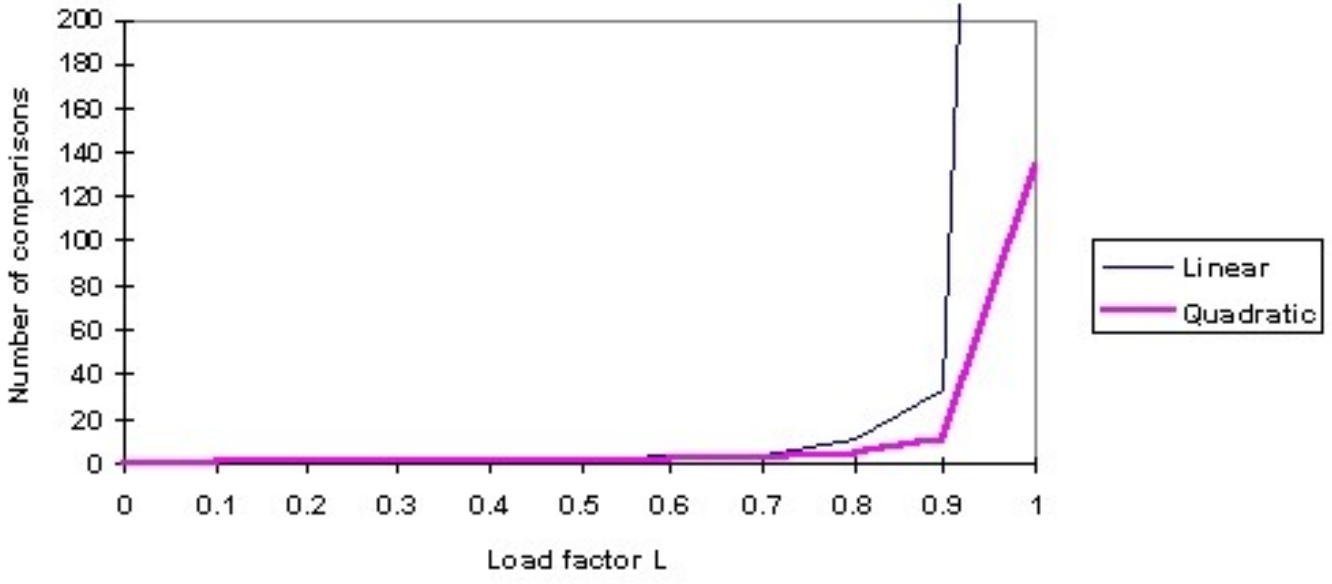
Søk der element ikke ble funnet

Lineær og kvadratisk probing: Sammenligning av effektivitet av søking

Konstant hashlengde lik 1000, varierende load factor



Søk der element ble funnet



Søk der element ikke ble funnet

Åpen adressering med rehashing

- Probing med rehashing*:
  - Bruk en *annen* og anderledes hashfunksjon for å finne neste indeks/probe ved kollisjoner
  - Hvis neste indeks også er opptatt, prøv med f.eks. 2 ganger ny hashverdi, deretter 3 ganger ny verdi etc.
- Gir ofte bedre spredning enn lineær og kvadratisk probing
- Kan løse opp både primære og sekundære clustere
- Animasjon

*: Også kalt «dataavhengig probing»

Åpen adressering med bruk av en randomgenerator

- Probing med en (pseudo) randomgenerator:
  - Hvis kollisjon, bruk hashverdien som *seed* i en randomgenerator og beregn nye indekser som en sekvens av «tilfeldige» tall (% hashlengde) inntil ledig plass funnet
- Fungerer fordi alle randomgeneratorer egentlig er deterministiske (pseudo random) og sekvensen av tilfeldige tall kan gjenskapes
- Kan også gi bedre spredning enn lineær og kvadratisk probing

Når hashtabellen blir full

- Med åpen adressering bør hashlengden økes når:
  - Load factor blir > 0.8, eller:
  - Vi ikke finner noen ledig indeks ved kollisjoner
- Vanlig å doble lengden av hashtabellen:
  - Vet at effektiviteten er garantert for load factor < 0.5
  - Velger alltid hashlengde lik nærmeste primtall
- Økning av hashlengden er en $O(n)$ operasjon:
  - Alle elementer *må* hashes på nytt for at de skal kunne finnes igjen med ny hashlengde

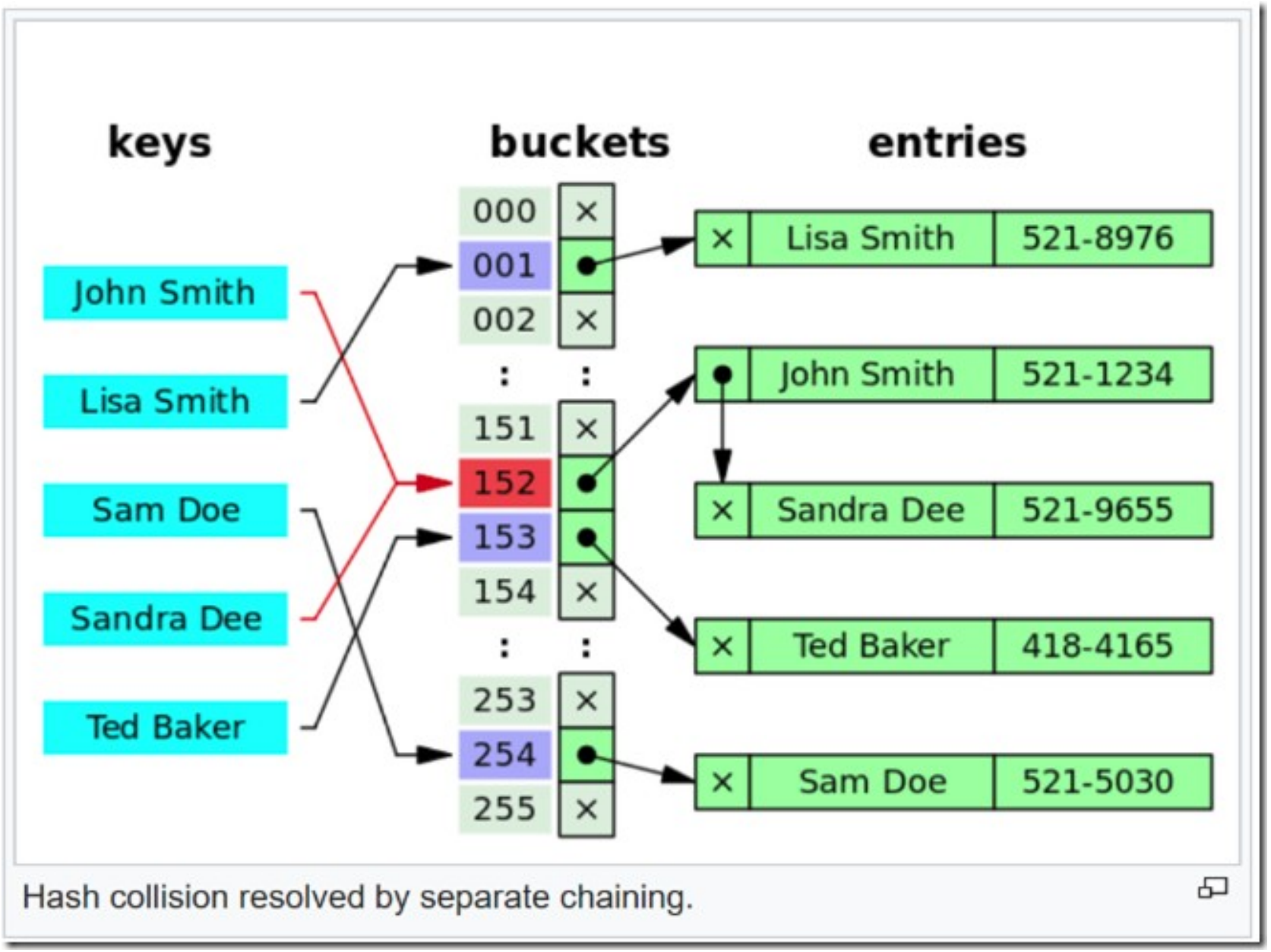
Fjerning av elementer fra en hashtabell med åpen adressering

- Problem: Vi risikerer å «bryte kjeden» ved å fjerne et element som ligger i en liste av probes
- Vanlig løsning:
  - *Ikke* fjern elementer, men bruk i stedet en ekstra boolsk array til å *merke* at elementer skal slettes
  - Ved kollisjoner kan vi stoppe kjeden av probes når vi kommer til et element som er merket som fjernet, og bare *overskrive* dette elementet
  - Alle elementer som er merket som fjernet, blir tatt vekk hver gang vi må gjøre en fullstendig rehashing i forbindelse med økning av lengden på hastabellen

Hashing med kjeding

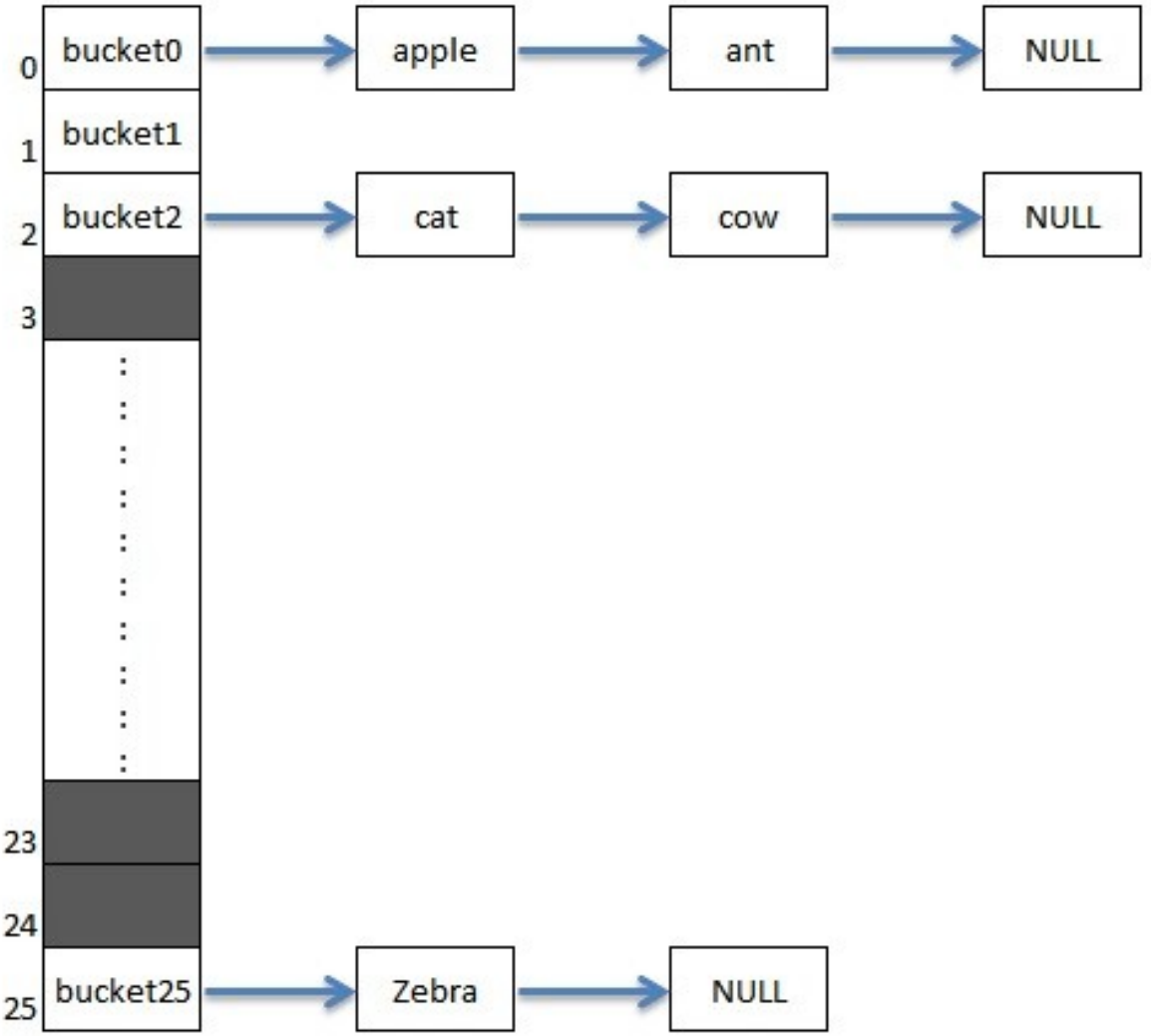
- Hashtabellen er en array av lister (buckets):
  - Vanlig å bruke usorterte lenkede lister
  - Alternativt kan listene «simuleres» ved å legge elementer som kolliderer i et «overflow» område i hashtabellen, for å spare overhead til pekere*
- Kollisjoner løses enkelt:
  - Alle elementer som får samme hashverdi legges inn i listen som ligger på denne indeksen i hashtabellen
  - Hashing med kjeding *partisjonerer* dataene opp i små delmengder som kan behandles effektivt

*: Se figur E.3 i Appendix E i læreboka



Hash collision resolved by separate chaining.

Kjeding: Eksempel



Animasjon av hashing med kjeding

- Eksempel med hashlengden lik 13
- Kan velge mellom hashing av heltall eller strenger
- Enkel hashfunksjon for heltall:

```
hash(key) = key % hash_lengde
```

- Demo: Open Hashing

Enkel implementasjon av kjeding

- Hashing av tekststrenger: `hashChained.java`
- Implementerer bare innsetting og søking i hashtabellen
- Ingen rehashing eller håndtering av høy load factor

- Koden skal brukes i siste del av obligatorisk oppgave 5:
  - Programmere metode for fjerning av en verdi fra hashtabellen

Kjeding: Fordeler og ulemper

- Fordeler:
  - Raskt, sammenligner bare elementer med lik hashverdi
  - Tåler load factor >> 1.0, mindre behov for økning av hashlengde og full rehashing
  - Fjerning av elementer er enkelt (lenket liste)
- Ulemper:
  - Krever $O(n)$ ekstra plass til referanser/pekere
  - Økning av hashlengde med full rehashing er mer komplisert/tidkrevende enn for åpen adressering, fordi vi må håndtere dynamisk hukommelse/pekere

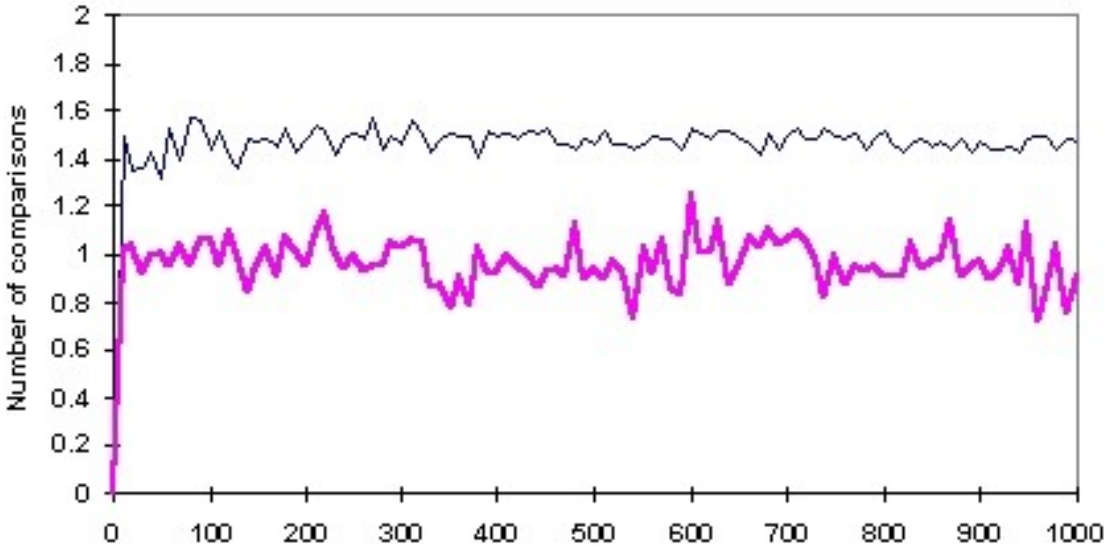
Effektivitet av hashing med kjeding

- Innsetting, søking og fjerning krever først én beregning av indeks i hashtabellen
- Innsetting vil alltid være $O(1)$ – kan sette inn nye dataelementer først i usortert liste
- Søking og fjerning krever et sekvensielt søk i den usorterte listen som ligger lagret på hashindeksen
- Hashing med kjeding blir derfor $O(1)$ hvis:
  - Hashfunksjonen sprer jevnt, slik at alle listene blir omtrent like lange
  - Load factor ikke er *for* stor, slik at listene ikke blir svært lange

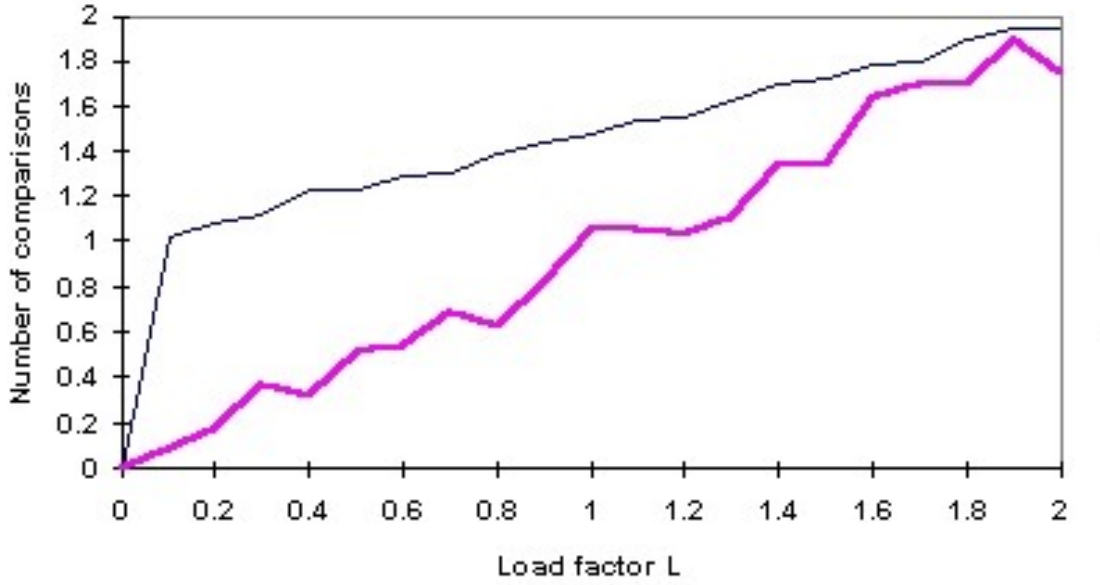
Load factor, hashlengde og kjeding

- Kjeding fungerer også for load factor større enn 1.0, hvis hashfunksjonen sprer jevnt
- Fra Wikipedia:
  - Chained hash tables remain effective even when the number of table entries *n* is much higher than the number of slots. Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

Effektivitet av kjeding: Test av søking



Load factor = 1.0
Varierende hashlengde



Fast hashlengde
Varierende load factor

Sammenligning: Søkealgoritmer/datastrukturer

Søke-algoritme	Data-struktur	Sortert?	Innsetting	Søking	Fjerning
Sekvensiell	Liste/array	Nei	O(1)	O(n)	O(n)
Binærsøk	Array	Ja	O(n)--	O(log n)	O(n)
Søketre*	Binært tre	Ja	O(log n)	O(log n)	O(log n)
Hashing**	Hashtabell	Tja	O(1)	O(1)	O(1)

Testprogram: Effektivitet av søketre vs. hashing

*: Garantert oppførsel (AVL) **: Ikke garantert