Hash Tables

Algorithms and Datastructures, F25, Lecture 7

Andreas Holck Høeg-Petersen

Department of Computer Science
Aalborg University

April 3, 2025

Opdateringer

- Næste programmeringsopgave er ude — har I alle set den?

Opdateringer

- Næste programmeringsopgave er ude — har I alle set den?
  - Der vil være lidt ekstra tid til exercises i dag, og det er blandt andet, så I eventuelt kan arbejde lidt med opgaven

Outline

Outline

Symbol tables

Associer en nøgle med en værdi

Et symbol table er en data struktur, der kan mappe fra en nøgle til en værdi (eller sattelit data).

Symbol tables

Associer en nøgle med en værdi

Et symbol table er en data struktur, der kan mappe fra en nøgle til en værdi (eller sattelit data).

- Vi har set dette med binære søgetræer, f.eks. i jeres programmeringsopgave, hvor I associere en dato med en person

Symbol tables

Associer en nøgle med en værdi

Et symbol table er en data struktur, der kan mappe fra en nøgle til en værdi (eller sattelit data).

- Vi har set dette med binære søgetræer, f.eks. i jeres programmeringsopgave, hvor I associere en dato med en person
- Et simpelt array kan faktisk også betragtes som et symbol table — men hvad er så nøglen, og hvad er værdien?

Symbol tables

Associer en nøgle med en værdi

Et symbol table er en data struktur, der kan mappe fra en nøgle til en værdi (eller sattelit data).

- Vi har set dette med binære søgetræer, f.eks. i jeres programmeringsopgave, hvor I associere en dato med en person
- Et simpelt array kan faktisk også betragtes som et symbol table — men hvad er så nøglen, og hvad er værdien?
  - Nøglen er indexet og værdien er elementet på det index

Symbol tables

Associer en nøgle med en værdi

Et symbol table er en data struktur, der kan mappe fra en nøgle til en værdi (eller sattelit data).

- Vi har set dette med binære søgetræer, f.eks. i jeres programmeringsopgave, hvor I associere en dato med en person
- Et simpelt array kan faktisk også betragtes som et symbol table — men hvad er så nøglen, og hvad er værdien?
  - Nøglen er indexet og værdien er elementet på det index
- Symbol tables — som også kaldes dictionaries — skal understøtte operationerne Insert, Search og Delete

Symbol tables

Associer en nøgle med en værdi

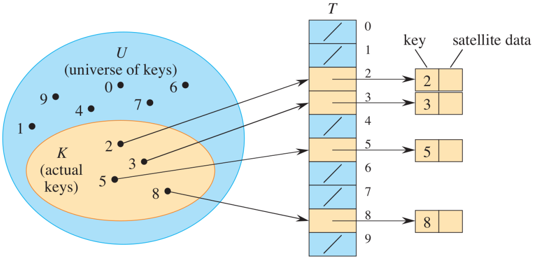
Et symbol table er en data struktur, der kan mappe fra en nøgle til en værdi (eller sattelit data).

- Vi har set dette med binære søgetræer, f.eks. i jeres programmeringsopgave, hvor I associere en dato med en person
- Et simpelt array kan faktisk også betragtes som et symbol table — men hvad er så nøglen, og hvad er værdien?
  - Nøglen er indexet og værdien er elementet på det index
- Symbol tables — som også kaldes dictionaries — skal understøtte operationerne Insert, Search og Delete
- Hash tables er en effektiv datastruktur til dette formål, der kan understøtte alle operationerne i $O(1)$ tid!

Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

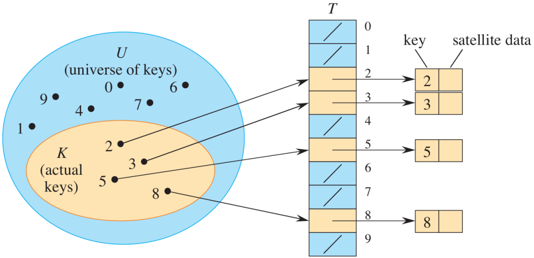


Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

- Vi har en applikation, hvor vi skal kunne opbevare elementer, der alle sammen har en unik nøgle associeret med dem

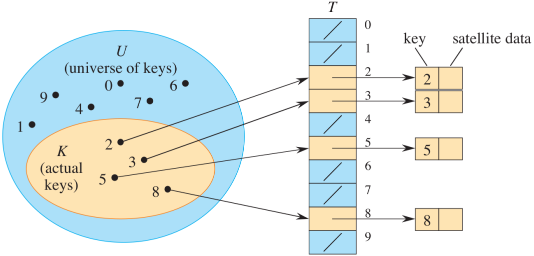


Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

- Vi har en applikation, hvor vi skal kunne opbevare elementer, der alle sammen har en unik nøgle associeret med dem
- Nøglerne stammer fra et univers af nøgler $U = \{0, 1, \ldots, m-1\}$

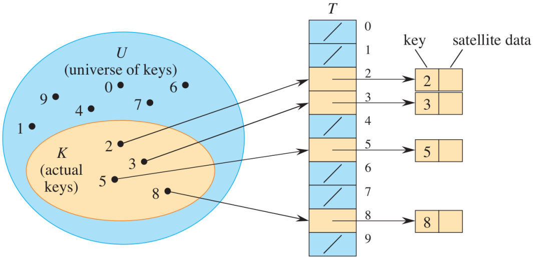


Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

- Vi har en applikation, hvor vi skal kunne opbevare elementer, der alle sammen har en unik nøgle associeret med dem
- Nøglerne stammer fra et univers af nøgler $U = \{0, 1, \ldots, m-1\}$
- Vi kan repræsentere vores tabel med et array $T[0:m-1]$, hvor vi har plads til m elementer

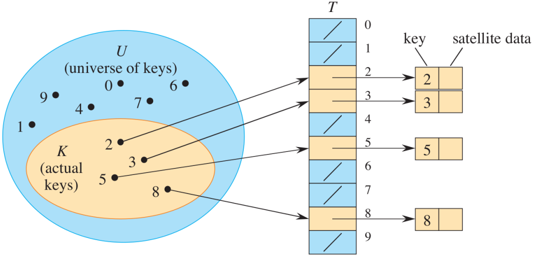


Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

- Vi har en applikation, hvor vi skal kunne opbevare elementer, der alle sammen har en unik nøgle associeret med dem
- Nøglerne stammer fra et univers af nøgler $U = \{0, 1, \ldots, m-1\}$
- Vi kan repræsentere vores tabel med et array $T[0:m-1]$, hvor vi har plads til m elementer
- Elementet med nøgle k finder vi så på plads $T[k]$ (hvor $T[k] = \text{NIL}$, hvis ikke elementet findes i T)

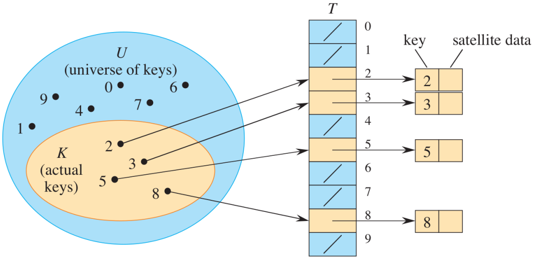


Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

- Vi har en applikation, hvor vi skal kunne opbevare elementer, der alle sammen har en unik nøgle associeret med dem
- Nøglerne stammer fra et univers af nøgler $U = \{0, 1, \ldots, m-1\}$
- Vi kan repræsentere vores tabel med et array $T[0:m-1]$, hvor vi har plads til m elementer
- Elementet med nøgle k finder vi så på plads $T[k]$ (hvor $T[k] = \text{NIL}$, hvis ikke elementet findes i T)
- Med m nøgler i U og n elementer i T er der altså $m - n$ tomme pladser i T

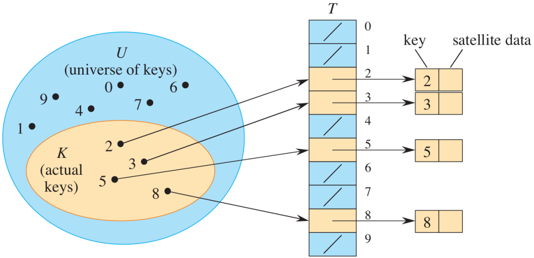


Direct-Address Tables

Basically bare et array

Den simpleste måde at implementere er bare med et array — dette kaldes en direct-address table.

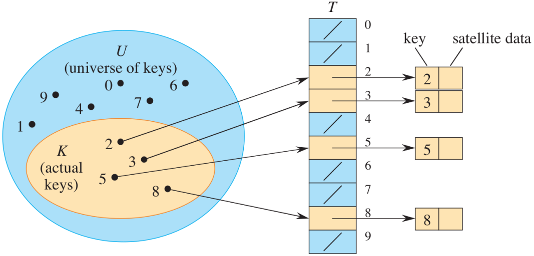
- Vi har en applikation, hvor vi skal kunne opbevare elementer, der alle sammen har en unik nøgle associeret med dem
- Nøglerne stammer fra et univers af nøgler $U = \{0, 1, \ldots, m-1\}$
- Vi kan repræsentere vores tabel med et array $T[0 : m-1]$, hvor vi har plads til m elementer
- Elementet med nøgle k finder vi så på plads $T[k]$ (hvor $T[k] = \text{NIL}$, hvis ikke elementet findes i T)
- Med m nøgler i U og n elementer i T er der altså $m - n$ tomme pladser i T
- Alle operationer er trivielle og kører i $O(1)$ tid



Direct-Address Tables

Eksempel

Eksempel

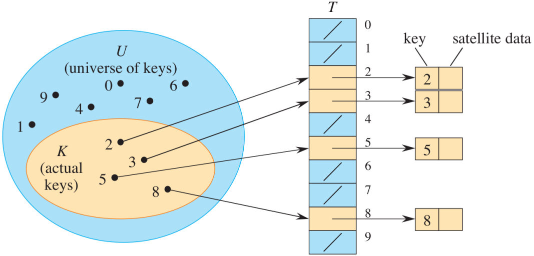


Direct-Address Tables

Eksempel

Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder

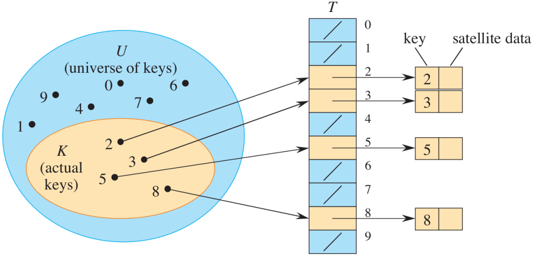


Direct-Address Tables

Eksempel

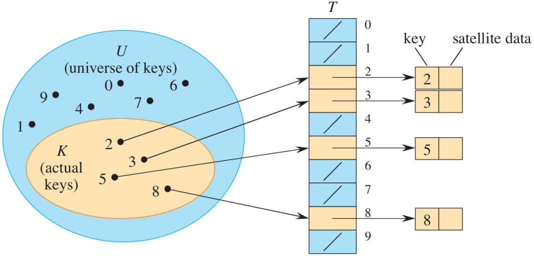
Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder
- Vi bruger en tabel, hvor ordene er nøgler og antal forekomster er værdierne



Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder
- Vi bruger en tabel, hvor ordene er nøgler og antal forekomster er værdierne
- Vi kan associere unikke nøgler til ordene ved at bruge deres position, når de er sorteret i alfabetisk rækkefølge

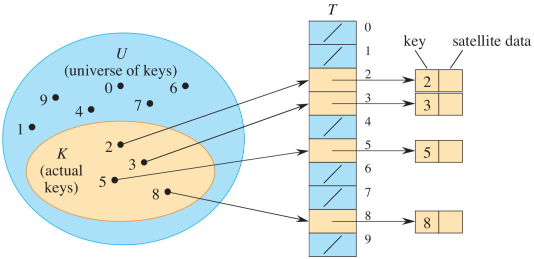


Direct-Address Tables

Eksempel

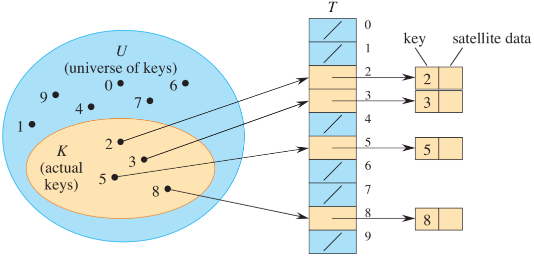
Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder
- Vi bruger en tabel, hvor ordene er nøgler og antal forekomster er værdierne
- Vi kan associere unikke nøgler til ordene ved at bruge deres position, når de er sorteret i alfabetisk rækkefølge
- Lad os sige, at der er 500.000 danske ord, så vi kunne have følgende:



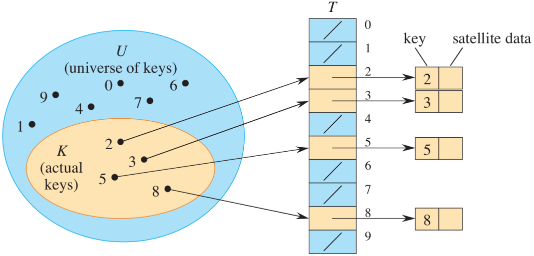
Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder
- Vi bruger en tabel, hvor ordene er nøgler og antal forekomster er værdierne
- Vi kan associere unikke nøgler til ordene ved at bruge deres position, når de er sorteret i alfabetisk rækkefølge
- Lad os sige, at der er 500.000 danske ord, så vi kunne have følgende:
  - 'Abe'.$key = 0$



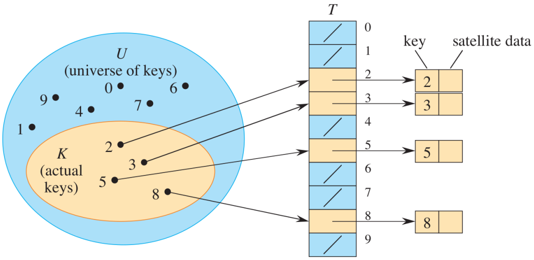
Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder
- Vi bruger en tabel, hvor ordene er nøgler og antal forekomster er værdierne
- Vi kan associere unikke nøgler til ordene ved at bruge deres position, når de er sorteret i alfabetisk rækkefølge
- Lad os sige, at der er 500.000 danske ord, så vi kunne have følgende:
    - ‘Abe’.$key = 0$
    - ‘Mark’.$key = 250.000$



Eksempel

- Vi har fået en tekst og vil gerne tælle, hvor mange gange hvert ord optræder
- Vi bruger en tabel, hvor ordene er nøgler og antal forekomster er værdierne
- Vi kan associere unikke nøgler til ordene ved at bruge deres position, når de er sorteret i alfabetisk rækkefølge
- Lad os sige, at der er 500.000 danske ord, så vi kunne have følgende:
  - ‘Abe’. $key = 0$
  - ‘Mark’. $key = 250.000$
  - ‘År’. $key = 499.999$

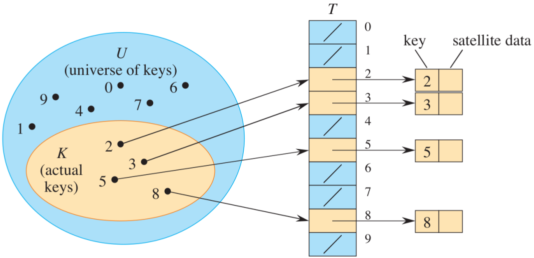


Direct-Address Tables

Eksempel

PROBLEM!!!

Eksempel

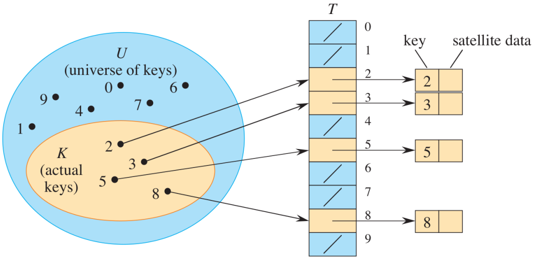

T
U
(universe of keys)
K
(actual
keys)
key
satellite data
0
1
2
3
4
5
6
7
8
9

Direct-Address Tables

Eksempel

Eksempel

- Vi møder måske kun 100-200 forskellige ord i artiklen, men vores tabel har plads til mere end 1000 gange så mange!

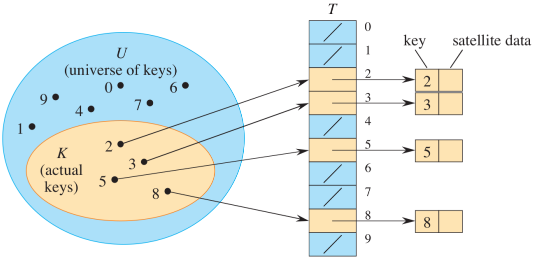


Direct-Address Tables

Eksempel

Eksempel

- Vi møder måske kun 100-200 forskellige ord i artiklen, men vores tabel har plads til mere end 1000 gange så mange!
- Direct-Address tables er ikke smarte, når U er langt større end K (sættet af aktuelle nøgler)

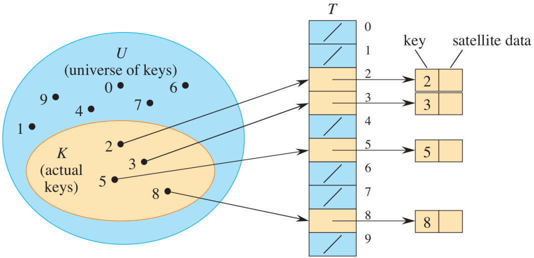


Direct-Address Tables

Eksempel

Eksempel

- Vi møder måske kun 100-200 forskellige ord i artiklen, men vores tabel har plads til mere end 1000 gange så mange!
- Direct-Address tables er ikke smarte, når U er langt større end K (sættet af aktuelle nøgler)
- Hvad værre er: vi kan have at U er så stort (eller uendeligt!), at det er umuligt at have en tabel med plads til alle nøgler

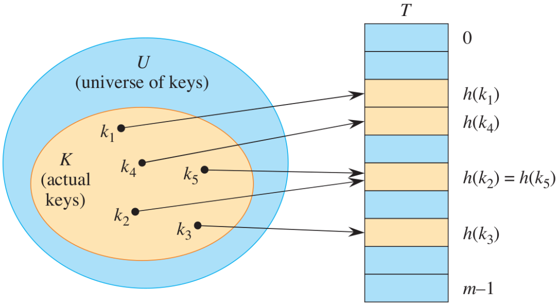


Outline

Hashing

Har ikke noget med weed at gøre

I stedet for at have plads til alle $|U|$ nøgler, kan vi bruge en hash funktion til at mappe fra en nøgle til en plads i et hash table.

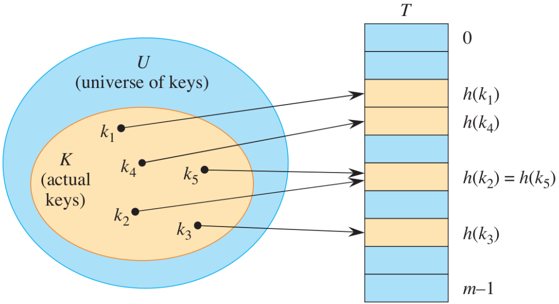


Hashing

Har ikke noget med weed at gøre

I stedet for at have plads til alle $|U|$ nøgler, kan vi bruge en hash funktion til at mappe fra en nøgle til en plads i et hash table.

- At 'hashe' betyder at skære eller hugge noget i stykker (biksemad!)

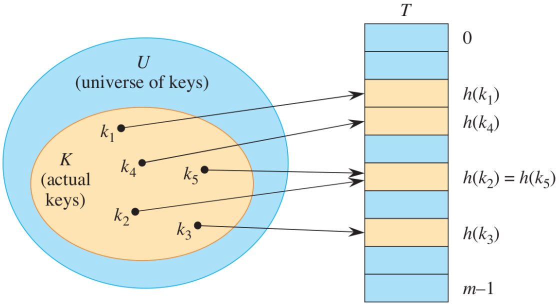


Hashing

Har ikke noget med weed at gøre

I stedet for at have plads til alle $|U|$ nøgler, kan vi bruge en hash funktion til at mappe fra en nøgle til en plads i et hash table.

- At 'hashe' betyder at skære eller hugge noget i stykker (biksemad!)
- En hash funktion $h : U \to \{0, 1, 2, \ldots, m-1\}$ er en funktion fra universet af nøgler til et tal mellem 0 og $m-1$

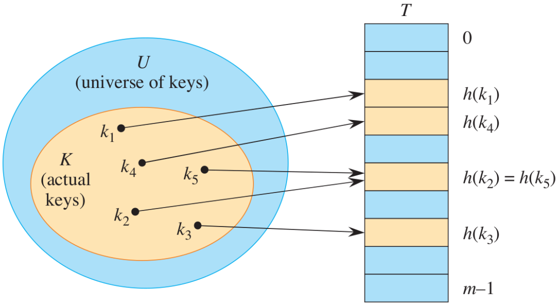


Hashing

Har ikke noget med weed at gøre

I stedet for at have plads til alle $|U|$ nøgler, kan vi bruge en hash funktion til at mappe fra en nøgle til en plads i et hash table.

- At 'hashe' betyder at skære eller hugge noget i stykker (biksemad!)
- En hash funktion $h : U \to \{0, 1, 2, \ldots, m-1\}$ er en funktion fra universet af nøgler til et tal mellem 0 og $m-1$
  - Eksempel: $h(k) = k \mod m$

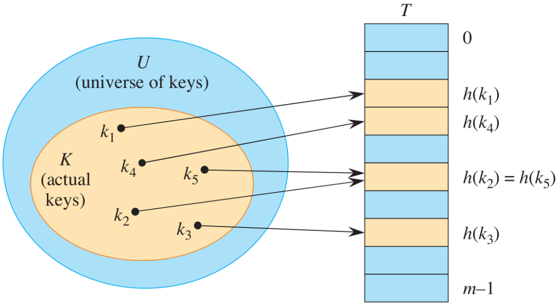


Hashing

Har ikke noget med weed at gøre

I stedet for at have plads til alle $|U|$ nøgler, kan vi bruge en hash funktion til at mappe fra en nøgle til en plads i et hash table.

- At 'hashe' betyder at skære eller hugge noget i stykker (biksemad!)
- En hash funktion $h : U \to \{0, 1, 2, \ldots, m-1\}$ er en funktion fra universet af nøgler til et tal mellem 0 og $m-1$
  - Eksempel: $h(k) = k \mod m$
- Vores hash table $T[0 : m-1]$ er et array med plads til m elementer, og vi antager at $m \ll |U|$ (altså at m er meget mindre end størrelsen på U)

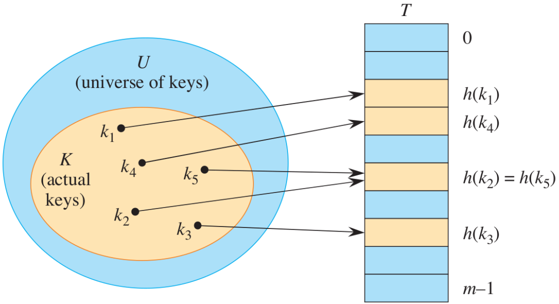


Hashing

Har ikke noget med weed at gøre

I stedet for at have plads til alle $|U|$ nøgler, kan vi bruge en hash funktion til at mappe fra en nøgle til en plads i et hash table.

- At 'hashe' betyder at skære eller hugge noget i stykker (biksemad!)
- En hash funktion $h : U \to \{0, 1, 2, \ldots, m-1\}$ er en funktion fra universet af nøgler til et tal mellem 0 og $m-1$
  - Eksempel: $h(k) = k \mod m$
- Vores hash table $T[0 : m-1]$ er et array med plads til m elementer, og vi antager at $m \ll |U|$ (altså at m er meget mindre end størrelsen på U)
- Dermed står elementet med nøgle k på plads $T[h(k)]$



Hashing

Problem. . . ?

Hashing

Kollisioner

- To (eller flere) nøgler kan risikere at hash til samme værdi, dvs. $h(k_1) = h(k_2)$

Hashing

Kollisioner

- To (eller flere) nøgler kan risikere at hash til samme værdi, dvs. $h(k_1) = h(k_2)$
- Dette skaber en såkaldt kollision, da begge nøgler nu vil skulle stå på den samme plads i T

Hashing

Kollisioner

- To (eller flere) nøgler kan risikere at hash til samme værdi, dvs. $h(k_1) = h(k_2)$
- Dette skaber en såkaldt kollision, da begge nøgler nu vil skulle stå på den samme plads i T
- Eftersom $m < |U|$, så kan kollisioner ikke undgåes

Hashing

Kollisioner

- To (eller flere) nøgler kan risikere at hash til samme værdi, dvs. $h(k_1) = h(k_2)$
- Dette skaber en såkaldt kollision, da begge nøgler nu vil skulle stå på den samme plads i T
- Eftersom $m < |U|$, så kan kollisioner ikke undgåes
- Derfor skal vi dels

- To (eller flere) nøgler kan risikere at hash til samme værdi, dvs. $h(k_1) = h(k_2)$
- Dette skaber en såkaldt kollision, da begge nøgler nu vil skulle stå på den samme plads i T
- Eftersom $m < |U|$, så kan kollisioner ikke undgåes
- Derfor skal vi dels
  1. Finde en teknik for at konstruere hash-funktioner, der reducerer antallet af kollisioner

Hashing

Kollisioner

- To (eller flere) nøgler kan risikere at hash til samme værdi, dvs. $h(k_1) = h(k_2)$
- Dette skaber en såkaldt kollision, da begge nøgler nu vil skulle stå på den samme plads i T
- Eftersom $m < |U|$, så kan kollisioner ikke undgåes
- Derfor skal vi dels
    1. Finde en teknik for at konstruere hash-funktioner, der reducerer antallet af kollisioner
    2. Finde en teknik til at håndtere kollisioner, når de uundgåeligt optræder

Hash-funktioner

Ideelt set

- Vi antager at alle nøgler er ikke-negative heltal - hvis ikke, kan vi altid lave dem om til dette (f.eks. tage ASCII-værdien af en streng)

Hash-funktioner

Ideelt set

- Vi antager at alle nøgler er ikke-negative heltal - hvis ikke, kan vi altid lave dem om til dette (f.eks. tage ASCII-værdien af en streng)
- Idealet er, at h er en independent uniform hash function

Hash-funktioner

Ideelt set

- Vi antager at alle nøgler er ikke-negative heltal - hvis ikke, kan vi altid lave dem om til dette (f.eks. tage ASCII-værdien af en streng)
- Idealet er, at h er en independent uniform hash function
  - independent vil sige, at outputtet ikke afhænger af hvad andre nøgler har hashet til

Hash-funktioner

Ideelt set

- Vi antager at alle nøgler er ikke-negative heltal - hvis ikke, kan vi altid lave dem om til dette (f.eks. tage ASCII-værdien af en streng)
- Idealet er, at h er en independent uniform hash function
  - independent vil sige, at outputtet ikke afhænger af hvad andre nøgler har hashet til
  - uniform vil sige, at alle m værdier i h's værdimængde er lige sandsynlige

Hash-funktioner

Ideelt set

- Vi antager at alle nøgler er ikke-negative heltal - hvis ikke, kan vi altid lave dem om til dette (f.eks. tage ASCII-værdien af en streng)
- Idealet er, at h er en independent uniform hash function
  - independent vil sige, at outputtet ikke afhænger af hvad andre nøgler har hashet til
  - uniform vil sige, at alle m værdier i h's værdimængde er lige sandsynlige
- For at kunne sikre dette er vi nødt til at vide noget om distributionen af nøgler — og det gør vi sjældent

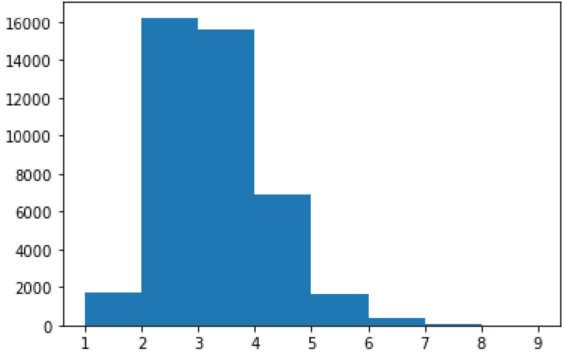


Figure: Antal vokaler i ord er en dårlig hash-funktion, der hverken er independent eller uniform

AALBORG UNIVERSITET

Static hashing

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
    - Divisionsmetoden: $h(k) = k \mod m$
        - ★ Hurtig og effektiv
        - ★ Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
    - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - ★ Hurtig og effektiv
    - ★ Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - ★ A er en konstant mellem 0 og 1
    - ★ m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Random hashing

- Vi definerer en familie $\mathcal{H}$ af hash-funktioner, der alle mapper U til $\{0, 1, \ldots, m-1\}$

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Random hashing

- Vi definerer en familie $\mathcal{H}$ af hash-funktioner, der alle mapper U til $\{0, 1, \ldots, m-1\}$
- Når vores program starter vælger vi en tilfældig hash-funktion $h \in \mathcal{H}$

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Random hashing

- Vi definerer en familie $\mathcal{H}$ af hash-funktioner, der alle mapper U til $\{0, 1, \ldots, m-1\}$
- Når vores program starter vælger vi en tilfældig hash-funktion $h \in \mathcal{H}$
- Dette beskytter imod, at en ondsindet agent kan finde n nøgler, der alle hasher til samme værdi

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Random hashing

- Vi definerer en familie $\mathcal{H}$ af hash-funktioner, der alle mapper U til $\{0, 1, \dots, m-1\}$
- Når vores program starter vælger vi en tilfældig hash-funktion $h \in \mathcal{H}$
- Dette beskytter imod, at en ondsindet agent kan finde n nøgler, der alle hasher til samme værdi
- En familie $\mathcal{H}$ af hash-funktioner kan defineres ved

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Random hashing

- Vi definerer en familie $\mathcal{H}$ af hash-funktioner, der alle mapper U til $\{0, 1, \ldots, m-1\}$
- Når vores program starter vælger vi en tilfældig hash-funktion $h \in \mathcal{H}$
- Dette beskytter imod, at en ondsindet agent kan finde n nøgler, der alle hasher til samme værdi
- En familie $\mathcal{H}$ af hash-funktioner kan defineres ved
  - Talteori — magi med primtal og modulo

Hash-funktioner

Metoder til at designe hash-funktioner

Static hashing

- Bruger den samme hash-funktion hver gang
- Her håber vi at kunne hashe på en måde, der er uafhængig af mønstre i input-dataen
- To simple udgaver
  - Divisionsmetoden: $h(k) = k \mod m$
    - Hurtig og effektiv
    - Kan være god hvis m er et primtal, der ikke er tæt på en 2'er-potens
  - Multiplikationsmetoden: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
    - A er en konstant mellem 0 og 1
    - m kan vælges uafhængigt af A, og fungerer godt som en 2'er-potens
- Dog, der er ingen garantier med static hashing

Random hashing

- Vi definerer en familie $\mathcal{H}$ af hash-funktioner, der alle mapper U til $\{0, 1, \ldots, m-1\}$
- Når vores program starter vælger vi en tilfældig hash-funktion $h \in \mathcal{H}$
- Dette beskytter imod, at en ondsindet agent kan finde n nøgler, der alle hasher til samme værdi
- En familie $\mathcal{H}$ af hash-funktioner kan defineres ved
  - Talteori — magi med primtal og modulo
  - Multiplikationsmetoden — en sofistikeret udgave af førnævnte teknik, som garenterer at sandsynligheden for kollision mellem to nøgler højst er $2/m$

Outline

Separate chaining

Håndtering af kollisioner

Når vi uundgåeligt støder ind i kollisioner, så skal vi have metoder til at håndtere dette.

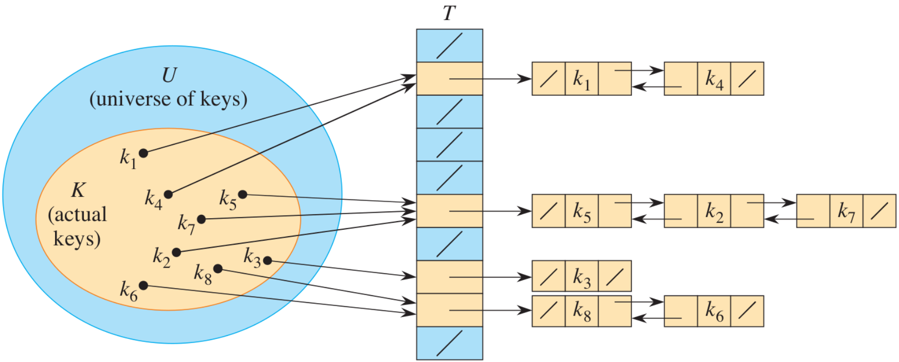
- Den mest simple metode kaldes chaining

Separate chaining

Håndtering af kollisioner

Når vi uundgåeligt støder ind i kollisioner, så skal vi have metoder til at håndtere dette.

- Den mest simple metode kaldes chaining
- Her gemmer vi en linked list på hver plads i T

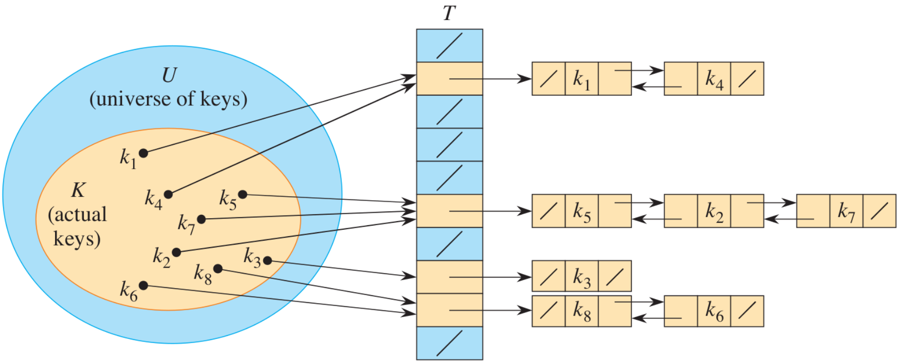


Separate chaining

Håndtering af kollisioner

Når vi uundgåeligt støder ind i kollisioner, så skal vi have metoder til at håndtere dette.

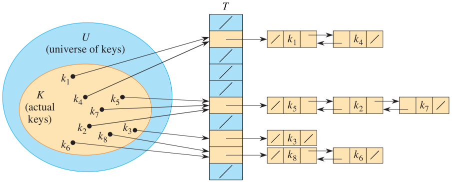
- Den mest simple metode kaldes chaining
- Her gemmer vi en linked list på hver plads i T
- Kollisioner løses således bare ved at tilføje elementet til listen



Chaining

Operation

Alle operationerne for dictionaries er nemme at implementere:

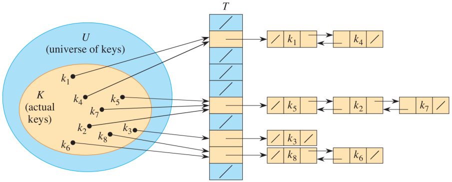


Chaining

Operation

Alle operationerne for dictionaries er nemme at implementere:

- Insert(T, x)
  - Indsæt i starten af listen på $T[h(x.key)]$
  - $O(1)$

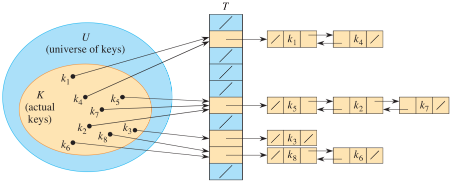


Chaining

Operation

Alle operationerne for dictionaries er nemme at implementere:

- Insert(T, x)
  - Indsæt i starten af listen på $T[h(x.key)]$
  - $O(1)$
- Delete(T, x)
  - Slet x fra listen på $T[h(x.key)]$
  - $O(1)$ (for doubly linked lists)

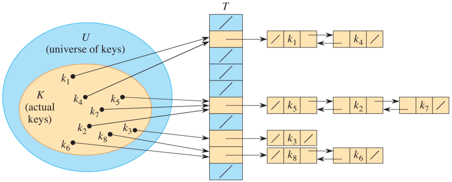


Chaining

Operation

Alle operationerne for dictionaries er nemme at implementere:

- Insert(T, x)
  - Indsæt i starten af listen på $T[h(x.key)]$
  - $O(1)$
- Delete(T, x)
  - Slet x fra listen på $T[h(x.key)]$
  - $O(1)$ (for doubly linked lists)
- Searc(T, k)
  - Søg efter k i listen på plads $T[h(k)]$
  - Lineær tid proportionelt med listens længde



Chaining

Kompleksitet

- Hvad er worst-case for Search i hash-table T med m pladser og n elementer (når vi bruger chaining)?

Chaining

Kompleksitet

- Hvad er worst-case for Search i hash-table T med m pladser og n elementer (når vi bruger chaining)?
  - Hvis alle nøgler hasher til samme værdi, så er det $\Theta(n)$

Chaining

Kompleksitet

- Hvad er worst-case for Search i hash-table T med m pladser og n elementer (når vi bruger chaining)?
  - Hvis alle nøgler hasher til samme værdi, så er det $\Theta(n)$
  - Tydeligvis bruger vi ikke hash tables for deres worst-case performance

Chaining

Kompleksitet

- Hvad er worst-case for Search i hash-table T med m pladser og n elementer (når vi bruger chaining)?
  - Hvis alle nøgler hasher til samme værdi, så er det $\Theta(n)$
  - Tydeligvis bruger vi ikke hash tables for deres worst-case performance
- I stedet, hvis vi antager independent uniform hashing kan vi definere $\alpha = n/m$ som vores load factor (det antal elementer, vi i gennemsnit forventer pr liste)

Chaining

Kompleksitet

- Hvad er worst-case for Search i hash-table T med m pladser og n elementer (når vi bruger chaining)?
  - Hvis alle nøgler hasher til samme værdi, så er det $\Theta(n)$
  - Tydeligvis bruger vi ikke hash tables for deres worst-case performance
- I stedet, hvis vi antager independent uniform hashing kan vi definere $\alpha = n/m$ som vores load factor (det antal elementer, vi i gennemsnit forventer pr liste)
- Nu kan man vise (se CLRS), at søgning i en hash table med chaining er givet ved $\Theta(1 + \alpha)$

Chaining

Kompleksitet

- Hvad er worst-case for Search i hash-table T med m pladser og n elementer (når vi bruger chaining)?
  - Hvis alle nøgler hasher til samme værdi, så er det $\Theta(n)$
  - Tydeligvis bruger vi ikke hash tables for deres worst-case performance
- I stedet, hvis vi antager independent uniform hashing kan vi definere $\alpha = n/m$ som vores load factor (det antal elementer, vi i gennemsnit forventer pr liste)
- Nu kan man vise (se CLRS), at søgning i en hash table med chaining er givet ved $\Theta(1 + \alpha)$
- Og hvis antallet af elementer i T er proportionel med antallet af pladser (ie. $n = O(m)$), så har vi ydermere $\alpha = n/m = O(m)/m = O(1)$, hvilket vil sige, at også søgning er $O(1)$ i gennemsnit

Outline

Exercises

Super fedt! <3

På Moodle! Go! Fungerer det fint?

Outline

Open addressing

Sig farvel til pointers

Et populært alternativt til chaining gemmer alle elementer direkte i tabellen og benytter probing til at løse kollisioner.

Open addressing

Sig farvel til pointers

Et populært alternativt til chaining gemmer alle elementer direkte i tabellen og benytter probing til at løse kollisioner.

- Hver plads i tabellen indeholder enten et element eller NIL

Open addressing

Sig farvel til pointers

Et populært alternativt til chaining gemmer alle elementer direkte i tabellen og benytter probing til at løse kollisioner.

- Hver plads i tabellen indeholder enten et element eller NIL
- Da tabellen dermed kan blive fyldt, vil $n \leq m$ hvormed vores load factor $\alpha \leq 1$

Open addressing

Sig farvel til pointers

Et populært alternativt til chaining gemmer alle elementer direkte i tabellen og benytter probing til at løse kollisioner.

- Hver plads i tabellen indeholder enten et element eller NIL
- Da tabellen dermed kan blive fyldt, vil $n \leq m$ hvormed vores load factor $\alpha \leq 1$
- Da vi undgår pointers, frigøres der hukommelse, der i stedet kan bruges til at have en større tabel og dermed kortere søge-tid

Open addressing

Sig farvel til pointers

Et populært alternativt til chaining gemmer alle elementer direkte i tabellen og benytter probing til at løse kollisioner.

- Hver plads i tabellen indeholder enten et element eller NIL
- Da tabellen dermed kan blive fyldt, vil $n \leq m$ hvormed vores load factor $\alpha \leq 1$
- Da vi undgår pointers, frigøres der hukommelse, der i stedet kan bruges til at have en større tabel og dermed kortere søge-tid
- Men hvad er probing så?

Open addressing
Probing

- Ideen er, at vi 'prober' tabellen for en ledig plads, et index af gangen

0	10
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Open addressing

Probing

- Ideen er, at vi 'prober' tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$

0	10
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Open addressing

Probing

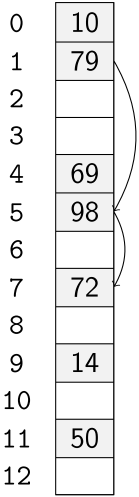
- Ideen er, at vi 'prober' tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$

0	10
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Open addressing

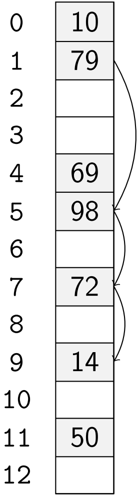
Probing

- Ideen er, at vi ‘prober’ tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$



Open addressing
Probing

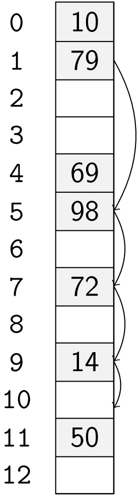
- Ideen er, at vi 'prober' tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$



Open addressing

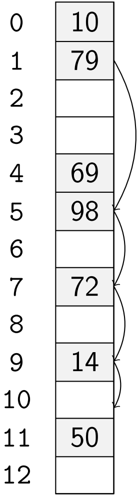
Probing

- Ideen er, at vi 'prober' tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$



Open addressing
Probing

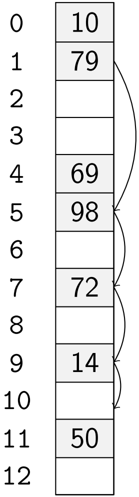
- Ideen er, at vi ‘prober’ tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$
- Vi udvider hash-funktionen til at inkludere probe-nummeret som input:
  - $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$



Open addressing

Probing

- Ideen er, at vi 'prober' tabellen for en ledig plads, et index af gangen
- En probe-sekvens er den rækkefølge, vi tjekker pladserne i tabellen i og er en permutation af $(0, 1, \ldots, m-1)$
- Vi udvider hash-funktionen til at inkludere probe-nummeret som input:
  - $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- Dermed bliver probe-sekevensen en tuple $\langle h(k, 0), h(k, 1), \ldots h(k, m-1) \rangle$



Hashing i open addressing

Hvordan hasher vi så?

- Ideelt set, så ønsker vi os independent uniform permutation hashing som betyder, at probe-sekvensen har lige stor sandsynlighed for genere en hvilken som helst af de $m!$ permutationer af $(0, 1, \ldots m-1)$
- Dette er dog meget svært at implementere (og mindre kan også gøre det)
- Vi ser på to teknikker, der generer hhv. m og m^2 probe-sekvenser:
  - Lineær probing
  - Dobbelt hashing

Lineær probing

AALBORG UNIVERSITET

Den simpleste metode er lineær probing:

$$h(k, i) = (h'(k) + i) \mod m$$

- Vi bruger en 'auxiliary' hash funktion $h' : U \rightarrow \{0, 1, \ldots, m-1\}$
- Probe-nummeret i bruges til at springe 1 plads frem hver gang
- Hvis $m = 8$ og $h'(k) = \lfloor k/3 \rfloor$ ville vi for $k = 10$ få probesekvensen $(3, 4, \ldots, 7, 0, 1, 2)$
- Nem at implementere, men kan kun generere m probe-sekvenser
- Har en tendens til at skabe primary clustering, ie. lange sekvenser hvor pladser er optaget

Dobbelt hashing

Faktisk er lineær probing et særtilfælde af en mere generel metode kaldet dobbelt hashing:

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

- Nu bruger vi 2 auxiliary hash funktioner, nemlig h_1 og h_2
- Vores første check (når $i = 0$) går til plads $T[h_1(k) \mod m]$ og derefter springer vi $h_2(k)$ pladser frem hver gang
- h_2 kan dermed ses som en slags 'step'-funktion
- For at garantere, at hele tabellen bliver undersøgt, så skal værdien af h_2 være et primtal relativt til m
  - Enten, lad m være en 2'er-potens og sørg for h_2 altid returnerer et ulige tal...
  - ... eller lad m være et primtal og sørg for at h_2 altid returnerer et tal mindre end m
- Dette giver m^2 forskellige probe-sekvenser, hvilket er tæt nok på idealet

0	26
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Figure: Eksempel på dobbelt hashing hvor $h_1(k) = k \mod 13$ og $h_2(k) = 1 + (k \mod 11)$

Insertion

Finally, some pseudo-kode!

Vi slutter med at se på koden for Insert, Search og. . . Delete?

Hash-Insert(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error 'hash table overflow'
```

Insertion

Finally, some pseudo-kode!

Vi slutter med at se på koden for Insert, Search og... Delete?

Hash-Insert(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0

Insertion

Finally, some pseudo-kode!

Vi slutter med at se på koden for Insert, Search og. . . Delete?

Hash-Insert(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q

Insertion

Finally, some pseudo-kode!

Vi slutter med at se på koden for Insert, Search og. . . Delete?

Hash-Insert(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så indsætter vi k og returnerer q

Insertion

Finally, some pseudo-kode!

Vi slutter med at se på koden for Insert, Search og. . . Delete?

Hash-Insert(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så indsætter vi k og returnerer q
- Ellers inkrementerer vi i og gentager indtil $i = m$

Insertion

Finally, some pseudo-kode!

Vi slutter med at se på koden for Insert, Search og. . . Delete?

Hash-Insert(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så indsætter vi k og returnerer q
- Ellers inkrementerer vi i og gentager indtil $i = m$
- Når vi ud af loopet, så er vores tabel helt fyldt

Search

Same same but different

Søgning er næsten identisk:

Hash-Search(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

Search

Same same but different

Søgning er næsten identisk:

Hash-Search(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0

Search

Same same but different

Søgning er næsten identisk:

Hash-Search(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q

Search

Same same but different

Søgning er næsten identisk:

Hash-Search(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så returnerer vi q

Search

Same same but different

Søgning er næsten identisk:

Hash-Search(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så returnerer vi q
- Ellers inkrementerer vi i...

Søgning er næsten identisk:

Hash-Search(T, k)

```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så returnerer vi q
- Ellers inkrementerer vi i...
- ... og gentager indtil vi finder en tom plads eller $i = m$

Search

Same same but different

Søgning er næsten identisk:

Hash-Search(T, k)

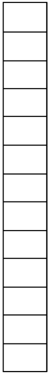
```
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        return q
    else i = i + 1
until T[q] == NIL or i == m
error 'hash table overflow'
```

- Vi starter med at initialisere proben til 0
- Så hasher vi nøglen og probe-nummeret og gemmer værdien i q
- Hvis $T[q]$ er tom, så returnerer vi q
- Ellers inkrementerer vi i...
- ... og gentager indtil vi finder en tom plads eller $i = m$
- Når vi ud af loopet, så findes k ikke i T

Deletion

Det må også være nemt!

Hvad så med deletion?

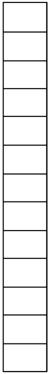


Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads

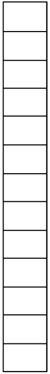


Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?



Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?
- Eksempel:

0	26
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?
- Eksempel:
  - Vi sletter nøglen '98'

0	26
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?
- Eksempel:
  - Vi sletter nøglen '98'
  - Nu skal vi søge efter '14', og vi tjekker først plads 1

0	26
1	79
2	
3	
4	69
5	
6	
7	72
8	
9	14
10	
11	50
12	

Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?
- Eksempel:
  - Vi sletter nøglen ‘98’
  - Nu skal vi søge efter ‘14’, og vi tjekker først plads 1
  - Der står hverken NIL eller ‘14’, så vi fortsætter

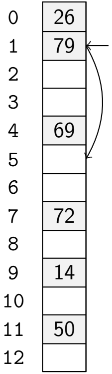
0	26
1	79 ←
2	
3	
4	69
5	
6	
7	72
8	
9	14
10	
11	50
12	

Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?
- Eksempel:
  - Vi sletter nøglen '98'
  - Nu skal vi søge efter '14', og vi tjekker først plads 1
  - Der står hverken NIL eller '14', så vi fortsætter
  - Nu tjekker vi plads 5, og der står NIL, fordi vi har slettet '98'



Deletion

Det må også være nemt!

Hvad så med deletion?

- Intuitivt: hvis vi vil slette et element, så indsætter vi bare NIL på dets plads
- Giver det nogen problemer?
- Eksempel:
    - Vi sletter nøglen ‘98’
    - Nu skal vi søge efter ‘14’, og vi tjekker først plads 1
    - Der står hverken NIL eller ‘14’, så vi fortsætter
    - Nu tjekker vi plads 5, og der står NIL, fordi vi har slettet ‘98’
    - Dermed returnerer vi NIL, selvom ‘14’ faktisk findes længere nede i tabellen!

0	26
1	79
2	
3	
4	69
5	
6	
7	72
8	
9	14
10	
11	50
12	

Deletion

Hva' gør vi så?

Det korte svar er, lad være at bruge open addressing, hvis deletion er nødvendigt.

- Vi kan ikke bare markere en plads med NIL
- En løsning er at bruge en anden speciel værdi DELETED
- Så skal vi bare modificere Hash-Insert til at behandle sådanne elementer som NIL
- Anses som problematisk grundet dårlig performance ved mange deletions
- Take-home message: Hvis man vil understøtte deletion er det bedre at bruge chaining

Tak for i dag!

Flere exercises..

Den bedste måde ikke at snyde sig selv på er lave exercises!