# Elemental Datastructures, Heaps and Priority Queues

## Algorithms and Datastructures, F25, Lecture 4

Andreas Holck Høeg-Petersen

Department of Computer Science
Aalborg University

March 6, 2025

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?

# Opdateringer



- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!

# Opdateringer



- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)
    - Vi kunne godt skrue ned på tiden brugt i forelæsningerne på at gennemgå algoritmerne, men...

# Opdateringer



- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)
    - Vi kunne godt skrue ned på tiden brugt i forelæsningerne på at gennemgå algoritmerne, men. . .
  - Flere mente tiden var godt fordelt mellem opgaver og forelæsning

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)
    - Vi kunne godt skrue ned på tiden brugt i forelæsningerne på at gennemgå algoritmerne, men...
  - Flere mente tiden var godt fordelt mellem opgaver og forelæsning
    - Der var også flere, der eksplicit var glade for den tid, vi bruger på at gennemgå eksempler og algoritmer

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)
    - Vi kunne godt skrue ned på tiden brugt i forelæsningerne på at gennemgå algoritmerne, men...
  - Flere mente tiden var godt fordelt mellem opgaver og forelæsning
    - Der var også flere, der eksplicit var glade for den tid, vi bruger på at gennemgå eksempler og algoritmer
  - 'Svært at holde fokus i del 2' + 'Fik ikke særligt meget at spise og drikke'

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)
    - Vi kunne godt skrue ned på tiden brugt i forelæsningerne på at gennemgå algoritmerne, men...
  - Flere mente tiden var godt fordelt mellem opgaver og forelæsning
    - Der var også flere, der eksplicit var glade for den tid, vi bruger på at gennemgå eksempler og algoritmer
  - 'Svært at holde fokus i del 2' + 'Fik ikke særligt meget at spise og drikke'
    - Del 2 vil være tungere pga tidspunktet

# Opdateringer

- Programmeringsopgave 1 — godt arbejde! Hvorfor var quicksort langsom for de store, random lister?
- Fra evaluering:
  - 20 svar sidst - fedt!
  - Flere ønsker mere tid til opgaver
    - Svært at gøre noget ved, da vi ikke har mere tid
    - Opgavesættene er ikke nødvendigvis designet til at kunne laves på 2 timer (dvs: hjemmearbejde nok nødvendigt, hvis man vil igennem dem!)
    - Vi kunne godt skrue ned på tiden brugt i forelæsningerne på at gennemgå algoritmerne, men...
  - Flere mente tiden var godt fordelt mellem opgaver og forelæsning
    - Der var også flere, der eksplicit var glade for den tid, vi bruger på at gennemgå eksempler og algoritmer
  - 'Svært at holde fokus i del 2' + 'Fik ikke særligt meget at spise og drikke'
    - Del 2 vil være tungere pga tidspunktet
    - Bedste løsning: få noget ordentligt at spise og drikke og husk at passe jeres søvn <3

# Outline


1. Elementære datastrukturer
2. Heaps
3. Exercises
4. Operationer på heaps
5. Heapsort
6. Priority Queues

# Outline


1. Elementære datastrukturer
2. Heaps
3. Exercises
4. Operationer på heaps
5. Heapsort
6. Priority Queues

# Datastrukturer

## Hvad og hvorfor?

En datastruktur er i bund og grund blot en struktureret samling af data.

# Datastrukturer

## Hvad og hvorfor?

En datastruktur er i bund og grund blot en struktureret samling af data.

- I kender allerede arrays, som er en sekvens af data-elementer af en bestemt type startende fra index 0 (eller 1, hvis man er CLRS-bogen. . . )

# Datastrukturer

## Hvad og hvorfor?

En datastruktur er i bund og grund blot en struktureret samling af data.

- I kender allerede arrays, som er en sekvens af data-elementer af en bestemt type startende fra index 0 (eller 1, hvis man er CLRS-bogen. . . )
- Vi benytter arrays som den fundamentale byggesten til at konstruere dynamiske mængder ('sets')

# Datastrukturer

## Hvad og hvorfor?

En datastruktur er i bund og grund blot en struktureret samling af data.

- I kender allerede arrays, som er en sekvens af data-elementer af en bestemt type startende fra index 0 (eller 1, hvis man er CLRS-bogen. . . )
- Vi benytter arrays som den fundamentale byggesten til at konstruere dynamiske mængder ('sets')
- Dynamiske mængder er noget, vi kan manipulere, f.eks. via Insert, Delete, Search eller lignende operationer

# Datastrukturer

## Hvad og hvorfor?

En datastruktur er i bund og grund blot en struktureret samling af data.

- I kender allerede arrays, som er en sekvens af data-elementer af en bestemt type startende fra index 0 (eller 1, hvis man er CLRS-bogen. . . )
- Vi benytter arrays som den fundamentale byggesten til at konstruere dynamiske mængder ('sets')
- Dynamiske mængder er noget, vi kan manipulere, f.eks. via Insert, Delete, Search eller lignende operationer
- Vi starter med at kigge på stacks og queues, som følger to forskellige principper for Insert og Delete

# Stacks

## Last in, first out

En helt fundamental datastruktur er stakken (en 'stack'). Den kan bedst sammenlignes med en stak tallerkener og følger LIFO-princippet: 'Last In, First Out'.

- Insertion og deletion kaldes henholdsvis Push og Pop
- Vi implementerer en stack med plads til $n$ elementer med et array $S[1 \dots n]$
- Vi definerer en attribut $S.top$, der peger på det index i $S$, hvor det seneste indsatte element befinder sig
- $S.size$ fortæller os hvor stor stacken er (dvs. $n$)

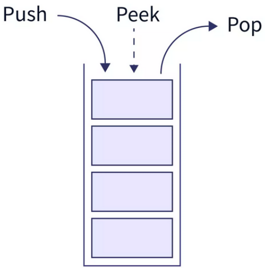



Figure: Source: `https://www.scaler.in/stack-operations/`

# Stacks

## Operationer

Stack-Empty($S$)

```
if S.top == 0
    return True
else return False
```

# Stacks

## Operationer

```
Stack-Empty(S)
if S.top == 0
    return True
else return False
```

```
Push(S, x)
if S.top == S.size
    error "overflow"
else
    S.top = S.top + 1
    S[S.top] = x
```

# Stacks

## Operationer

```
Stack-Empty(S)
if S.top == 0
    return True
else return False
```

```
Push(S, x)
if S.top == S.size
    error "overflow"
else
    S.top = S.top + 1
    S[S.top] = x
```

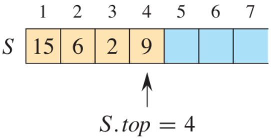
```
Pop(S)
if Stack-Empty(S)
    error "underflow"
else
    S.top = S.top − 1
    return S[S.top + 1]
```

# Stacks

## Operationer

Eksempel:

- Vi har stacken $S$ med $S.top == 4$




```
Push(S, x)
if S.top == S.size
    error "overflow"
else
    S.top = S.top + 1
    S[S.top] = x
```

```
Pop(S)
if Stack-Empty(S)
    error "underflow"
else
    S.top = S.top − 1
    return S[S.top + 1]
```

# Stacks

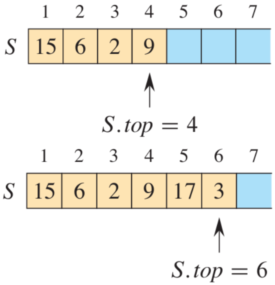
## Operationer

```
Push(S, x)
if S.top == S.size
    error "overflow"
else
    S.top = S.top + 1
    S[S.top] = x
```

```
Pop(S)
if Stack-Empty(S)
    error "underflow"
else
    S.top = S.top − 1
    return S[S.top + 1]
```

Eksempel:

- Vi har stacken $S$ med $S.top == 4$
- Vi kalder $\text{Push}(S, 17)$ og $\text{Push}(S, 3)$



# Stacks

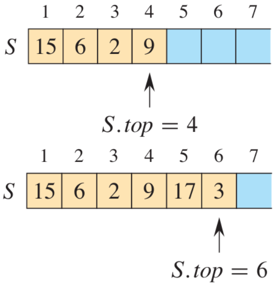
## Operationer

```
Push(S, x)
if S.top == S.size
        error "overflow"
else
        S.top = S.top + 1
        S[S.top] = x
```

```
Pop(S)
if Stack-Empty(S)
        error "underflow"
else
        S.top = S.top − 1
        return S[S.top + 1]
```

Eksempel:

- Vi har stacken $S$ med $S.top == 4$
- Vi kalder Push$(S, 17)$ og Push$(S, 3)$
- Nu har vi $S.top == 6$



# Stacks

## Operationer

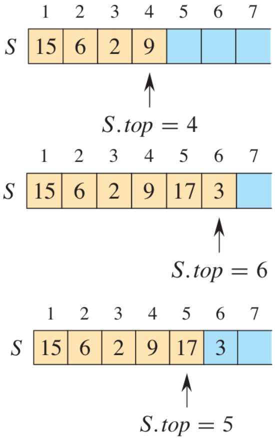


```
Push(S, x)
if S.top == S.size
        error "overflow"
else
        S.top = S.top + 1
        S[S.top] = x
```

```
Pop(S)
if Stack-Empty(S)
        error "underflow"
else
        S.top = S.top − 1
        return S[S.top + 1]
```

Eksempel:

- Vi har stacken $S$ med $S.top == 4$
- Vi kalder Push$(S, 17)$ og Push$(S, 3)$
- Nu har vi $S.top == 6$
- Vi kalder Pop$(S)$



# Stacks

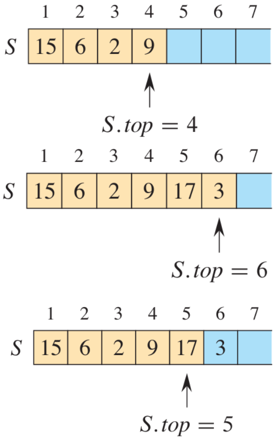
## Operationer

```
Push(S, x)
if S.top == S.size
        error "overflow"
else
        S.top = S.top + 1
        S[S.top] = x
```

```
Pop(S)
if Stack-Empty(S)
        error "underflow"
else
        S.top = S.top − 1
        return S[S.top + 1]
```

Eksempel:

- Vi har stacken $S$ med $S.top == 4$
- Vi kalder Push$(S, 17)$ og Push$(S, 3)$
- Nu har vi $S.top == 6$
- Vi kalder Pop$(S)$
- Kaldet returnerer 3 og sætter $S.top = 5$



# Stacks

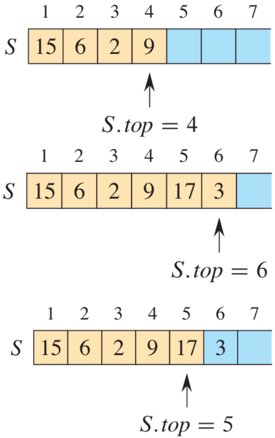
## Operationer

```
Push(S, x)
if S.top == S.size
        error "overflow"
else
        S.top = S.top + 1
        S[S.top] = x
```

```
Pop(S)
if Stack-Empty(S)
        error "underflow"
else
        S.top = S.top − 1
        return S[S.top + 1]
```

Eksempel:

- Vi har stacken $S$ med $S.top == 4$
- Vi kalder Push$(S, 17)$ og Push$(S, 3)$
- Nu har vi $S.top == 6$
- Vi kalder Pop$(S)$
- Kaldet returnerer 3 og sætter $S.top = 5$
- Bemærk at elementet stadig er i arrayet!



# Queues

## First in, first out

En anden klassisk datastruktur er en queue, altså en kø. Her følger vi FIFO-princippet: 'First In, First Out'.

- Insertion og deletion kalder vi henholdsvis Enqueue og Dequeue
- For en queue med $n - 1$ pladser bruger vi et array $Q[1 : n]$
- $Q.head$ angiver indexet på det 'forreste' element i køen, men $Q.tail$ angiver, hvor det næste element skal indsættes
- Vi implementerer vores queue med wrap-around: dvs. hvis $Q.tail == Q.size$ efter vi har indsat et element sætter vi $Q.tail = 1$

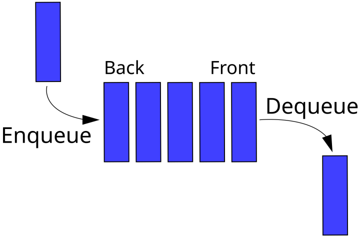



Figure: Source: Wikipedia

# Queues

## Operationer

ENQUEUE$(Q, x)$

$Q[Q.tail] = x$
**if** $Q.tail == Q.size$
  $Q.tail = 1$
**else** $Q.tail = Q.tail + 1$

# Queues

## Operationer

```
Enqueue(Q, x)
Q[Q.tail] = x
if Q.tail == Q.size
        Q.tail = 1
else Q.tail = Q.tail + 1
```

```
Dequeue(Q)
x = Q[Q.head]
if Q.head == Q.size
        Q.head = 1
else Q.head = Q.head + 1
return x
```

# Queues

## Operationer

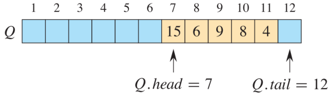
Enqueue($Q, x$)

```
Q[Q.tail] = x
if Q.tail == Q.size
        Q.tail = 1
else Q.tail = Q.tail + 1
```

Dequeue($Q$)

```
x = Q[Q.head]
if Q.head == Q.size
        Q.head = 1
else Q.head = Q.head + 1
return x
```

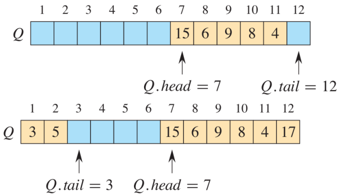
- Vi har en queue med 5 elementer, hvor $Q.head$ peger på indeks 7 og $Q.tail$ på indeks 12



```
Enqueue(Q, x)
Q[Q.tail] = x
if Q.tail == Q.size
      Q.tail = 1
else Q.tail = Q.tail + 1
```

```
Dequeue(Q)
x = Q[Q.head]
if Q.head == Q.size
      Q.head = 1
else Q.head = Q.head + 1
return x
```

- Vi har en queue med 5 elementer, hvor $Q.head$ peger på indeks 7 og $Q.tail$ på indeks 12
- Nu kalder vi Enqueue$(Q, 17)$, Enqueue$(Q, 3)$ og Enqueue$(Q, 5)$ (bemærk wrap-around!)



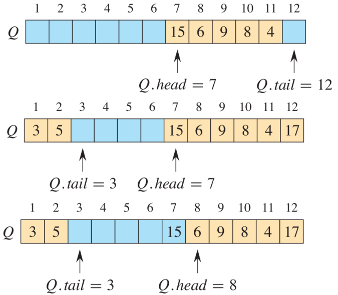
# Queues

## Operationer

```
Enqueue(Q, x)
Q[Q.tail] = x
if Q.tail == Q.size
        Q.tail = 1
else Q.tail = Q.tail + 1
```

```
Dequeue(Q)
x = Q[Q.head]
if Q.head == Q.size
        Q.head = 1
else Q.head = Q.head + 1
return x
```

- Vi har en queue med 5 elementer, hvor $Q.head$ peger på indeks 7 og $Q.tail$ på indeks 12
- Nu kalder vi Enqueue$(Q, 17)$, Enqueue$(Q, 3)$ og Enqueue$(Q, 5)$ (bemærk wrap-around!)
- Endelig kalder vi Dequeue$(Q)$ som returnerer værdien 15 og inkrementerer $Q.head$

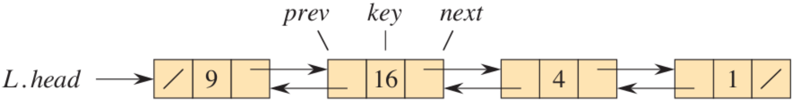


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:

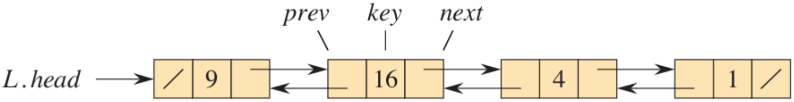


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet

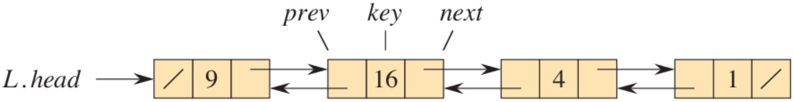


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet
  - $x.next$ er en pointer til næste element i listen

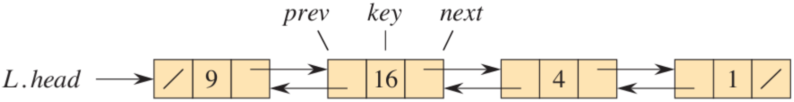


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet
  - $x.next$ er en pointer til næste element i listen
  - $x.prev$ er en pointer til forrige element i listen (kun for doubly linked lists)

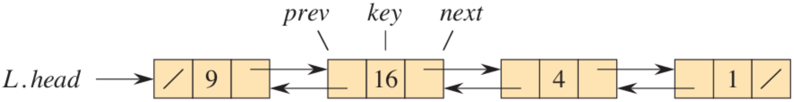


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet
  - $x.next$ er en pointer til næste element i listen
  - $x.prev$ er en pointer til forrige element i listen (kun for doubly linked lists)
- Selve listen har en enkelt attribut $L.head$ som peger på første element i listen

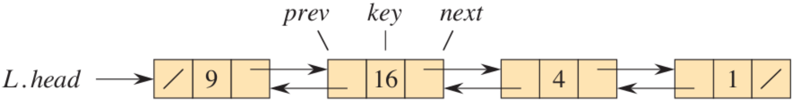


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet
  - $x.next$ er en pointer til næste element i listen
  - $x.prev$ er en pointer til forrige element i listen (kun for doubly linked lists)
- Selve listen har en enkelt attribut $L.head$ som peger på første element i listen
- Hvis $L.head == \text{nil}$ er listen tom

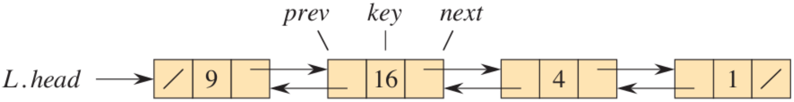


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet
  - $x.next$ er en pointer til næste element i listen
  - $x.prev$ er en pointer til forrige element i listen (kun for doubly linked lists)
- Selve listen har en enkelt attribut $L.head$ som peger på første element i listen
- Hvis $L.head ==$ nil er listen tom
- Hvis $x.next ==$ nil er $x$ sidste element i listen

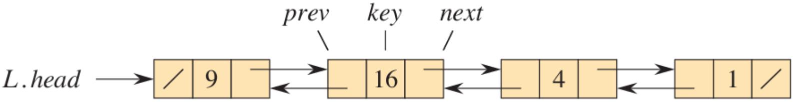


# Linked lists

## 'Hægtede' lister!

Den sidste af de basale datastruktur for nu er såkalde linked lists. Dette er en måde at implemntere dynamiske sekvenser på uden brug af arrays.

- Hvert element i en linked list $L$ er et objekt $x$ med følgende attributter:
  - $x.key$ indeholder selve værdien af elementet
  - $x.next$ er en pointer til næste element i listen
  - $x.prev$ er en pointer til forrige element i listen (kun for doubly linked lists)
- Selve listen har en enkelt attribut $L.head$ som peger på første element i listen
- Hvis $L.head ==$ nil er listen tom
- Hvis $x.next ==$ nil er $x$ sidste element i listen
- Bemærk at vi ikke behøver beslutte på forhånd, hvor stor en linked liste skal være (modsat et array)!



# Linked lists

## Operationer

**List-Search$(L, k)$**

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

- List-Search tager en liste $L$ og en værdi $k$ som input

# Linked lists

## Operationer

**List-Search**$(L, k)$

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

- List-Search tager en liste $L$ og en værdi $k$ som input
- Finder det første element i listen med nøglen $k$

# Linked lists

## Operationer

**List-Search($L$, $k$)**

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

- List-Search tager en liste $L$ og en værdi $k$ som input
- Finder det første element i listen med nøglen $k$
- Returnerer en pointer til dette element (NIL hvis der ikke findes sådan et element)

# Linked lists

## Operationer

**List-Search($L$, $k$)**

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

- List-Search tager en liste $L$ og en værdi $k$ som input
- Finder det første element i listen med nøglen $k$
- Returnerer en pointer til dette element (NIL hvis der ikke findes sådan et element)
- Hvilken worst-case køretid har List-Search?

# Linked lists

## Operationer

**List-Search($L$, $k$)**

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

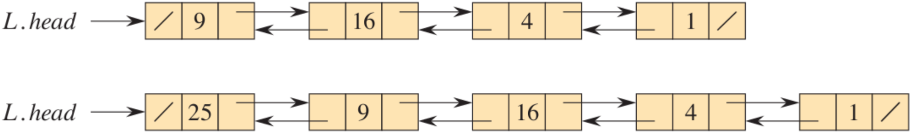
- List-Search tager en liste $L$ og en værdi $k$ som input
- Finder det første element i listen med nøglen $k$
- Returnerer en pointer til dette element (NIL hvis der ikke findes sådan et element)
- Hvilken worst-case køretid har List-Search?
  - Da vi i værste fald skal alle $n$ elementer igennem er køretiden $\Theta(n)$

AALBORG UNIVERSITET

**List-Prepend($L, x$)**

```
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
        L.head.prev = x
L.head = x
```

- List-Prepend tager en liste $L$ og et element $x$ som input



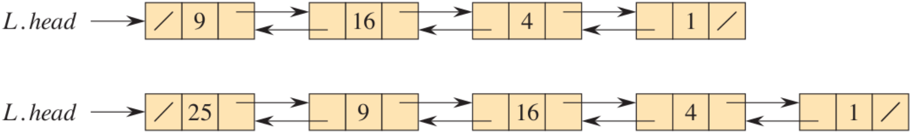
# Linked lists

## Prepend og insert

**List-Prepend($L, x$)**

$x.next = L.head$
$x.prev =$ NIL
**if** $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$

- List-Prepend tager en liste $L$ og et element $x$ som input
- Elementet $x$ indsættes som det første i listen



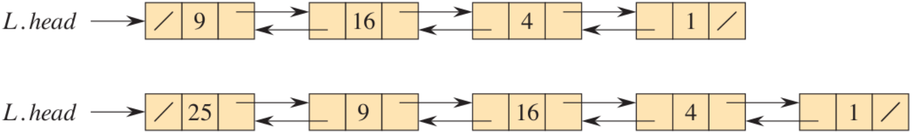
# Linked lists

## Prepend og insert

**List-Prepend($L, x$)**

```
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
        L.head.prev = x
L.head = x
```

- List-Prepend tager en liste $L$ og et element $x$ som input
- Elementet $x$ indsættes som det første i listen
- $L.head$ skal opdateres og pointers fra det gamle $L.head$ skal overføres til $x$



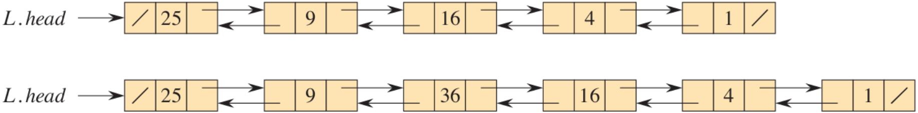
# Linked lists

## Prepend og insert

**List-Insert($x, y$)**

$x.next = y.next$
$x.prev = y$
**if** $y.next \neq$ NIL
    $y.next.prev = x$
$y.next = x$

- List-Insert tager to elementer $x$ og $y$ som input



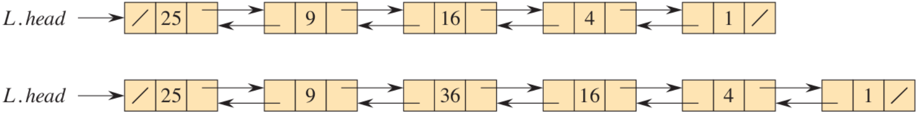
# Linked lists

## Prepend og insert

**List-Insert($x, y$)**

$x.next = y.next$
$x.prev = y$
**if** $y.next \neq$ NIL
    $y.next.prev = x$
$y.next = x$

- List-Insert tager to elementer $x$ og $y$ som input
- Element $x$ indsættes umiddelbart efter $y$ i listen



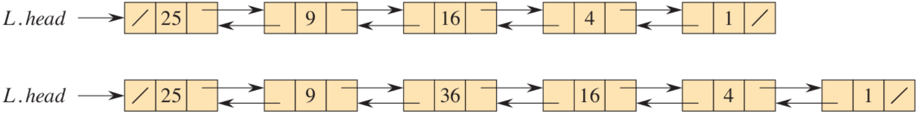
# Linked lists

## Prepend og insert

### List-Insert$(x, y)$

```
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

- List-Insert tager to elementer $x$ og $y$ som input
- Element $x$ indsættes umiddelbart efter $y$ i listen
- Pointers fra $y$ skal overføres til $x$, og $x.next$ skal pege på $y$



# Linked lists

## Prepend og insert

### List-Prepend($L, x$)

$x.next = L.head$
$x.prev =$ NIL
**if** $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$

### List-Insert($x, y$)

$x.next = y.next$
$x.prev = y$
**if** $y.next \neq$ NIL
    $y.next.prev = x$
$y.next = x$

- Bemærk at begge procedurer bevarer den interne rækkefølge af de eksisterende elementer

# Linked lists

## Prepend og insert

### List-Prepend($L, x$)

```
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
    L.head.prev = x
L.head = x
```

### List-Insert($x, y$)

```
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

- Bemærk at begge procedurer bevarer den interne rækkefølge af de eksisterende elementer
- Hvilken køretid har de to funktioner?

# Linked lists

## Prepend og insert

### List-Prepend($L, x$)

$x.next = L.head$
$x.prev =$ NIL
**if** $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$

### List-Insert($x, y$)

$x.next = y.next$
$x.prev = y$
**if** $y.next \neq$ NIL
    $y.next.prev = x$
$y.next = x$

- Bemærk at begge procedurer bevarer den interne rækkefølge af de eksisterende elementer
- Hvilken køretid har de to funktioner?
  - Begge er konstante $\Theta(1)$ operationer, da vi kun skal opdatere pointers

# Linked lists

## Prepend og insert

### List-Prepend$(L, x)$

$x.next = L.head$
$x.prev = \text{NIL}$
**if** $L.head \neq \text{NIL}$
  $L.head.prev = x$
$L.head = x$

### List-Insert$(x, y)$

$x.next = y.next$
$x.prev = y$
**if** $y.next \neq \text{NIL}$
  $y.next.prev = x$
$y.next = x$

- Bemærk at begge procedurer bevarer den interne rækkefølge af de eksisterende elementer
- Hvilken køretid har de to funktioner?
  - Begge er konstante $\Theta(1)$ operationer, da vi kun skal opdatere pointers
- Hvad ville det kræve at implementere samme funktionalitet i et array?

# Linked lists

## Prepend og insert

### List-Prepend$(L, x)$

```
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
    L.head.prev = x
L.head = x
```

### List-Insert$(x, y)$

```
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

- Bemærk at begge procedurer bevarer den interne rækkefølge af de eksisterende elementer
- Hvilken køretid har de to funktioner?
  - Begge er konstante $\Theta(1)$ operationer, da vi kun skal opdatere pointers
- Hvad ville det kræve at implementere samme funktionalitet i et array?
  - Alle elementer efter det indsatte ville skulle flyttes en plads 'opad' — $\Theta(n)$ i worst case!

# Linked lists

## Prepend og insert

### List-Prepend($L, x$)

$x.next = L.head$
$x.prev = \text{NIL}$
**if** $L.head \neq \text{NIL}$
    $L.head.prev = x$
$L.head = x$

### List-Insert($x, y$)

$x.next = y.next$
$x.prev = y$
**if** $y.next \neq \text{NIL}$
    $y.next.prev = x$
$y.next = x$

- Bemærk at begge procedurer bevarer den interne rækkefølge af de eksisterende elementer
- Hvilken køretid har de to funktioner?
  - Begge er konstante $\Theta(1)$ operationer, da vi kun skal opdatere pointers
- Hvad ville det kræve at implementere samme funktionalitet i et array?
  - Alle elementer efter det indsatte ville skulle flyttes en plads 'opad' — $\Theta(n)$ i worst case!
  - Og måske ville vi være nødt til at kopiere alt over i et nyt, større array

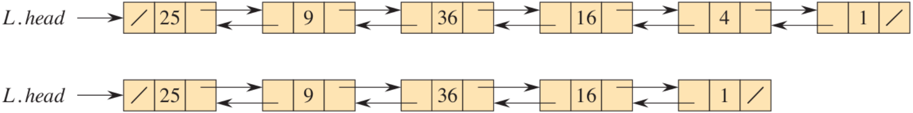
# Linked lists

## Deletion

**List-Delete($L, x$)**

**if** $x.next \neq$ NIL
    $x.prev.next = x.next$
**else** $L.head = x.next$
**if** $x.next \neq$ NIL
    $x.next.prev = x.next$

- List-Delete tager en liste $L$ og et element $x$ som input



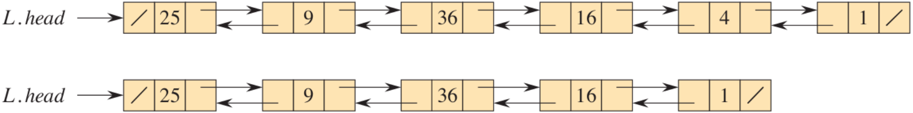
# Linked lists

## Deletion

### List-Delete($L, x$)

```
if x.next ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.next
```

- List-Delete tager en liste $L$ og et element $x$ som input
- Sletter $x$ fra listen og opdaterer pointers



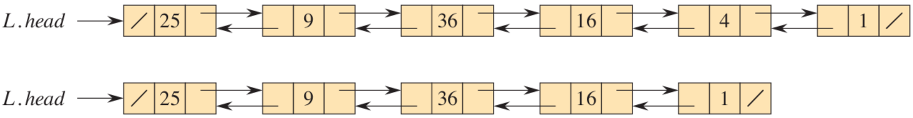
# Linked lists

## Deletion

**List-Delete($L, x$)**

**if** $x.next \neq$ NIL
  $x.prev.next = x.next$
**else** $L.head = x.next$
**if** $x.next \neq$ NIL
  $x.next.prev = x.next$

- List-Delete tager en liste $L$ og et element $x$ som input
- Sletter $x$ fra listen og opdaterer pointers
- Kører også i $\Theta(1)$



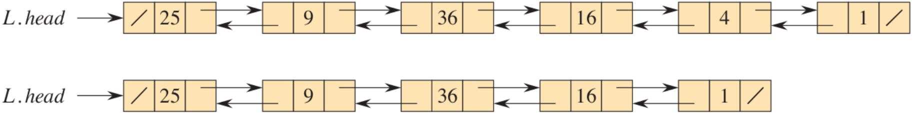
# Linked lists

## Deletion

### List-Delete($L, x$)

```
if x.next ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.next
```

- List-Delete tager en liste $L$ og et element $x$ som input
- Sletter $x$ fra listen og opdaterer pointers
- Kører også i $\Theta(1)$
  - Medmindre man lige skal finde elementet først...



# Outline

1 Elementære datastrukturer

2 Heaps

3 Exercises

4 Operationer på heaps

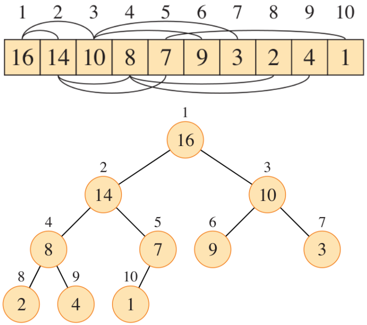
5 Heapsort

6 Priority Queues

## En lidt mindre basal datastruktur

Nu hvor vi har set en række meget simple datastrukturer kaster vi os over en marginalt mere avanceret størrelse: et (binært) heap.

- Vi repræsenterer et heap med plads til $n$ elementer med et array $A[1:n]$, men det egentlige antal elementer angives med attributten $A.\mathit{heap\text{-}size}$

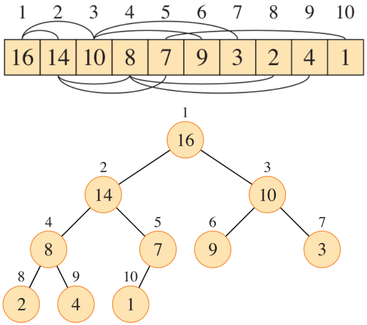


# Heaps

## En lidt mindre basal datastruktur

Nu hvor vi har set en række meget simple datastrukturer kaster vi os over en marginalt mere avanceret størrelse: et (binært) heap.

- Vi repræsenterer et heap med plads til $n$ elementer med et array $A[1:n]$, men det egentlige antal elementer angives med attributten $A.\mathit{heap\text{-}size}$
- Et heap kan ses som et næsten komplet træ, hvor alle niveauer er fyldt ud, pånær måske det sidste som dog skal være fyldt ud fra 'venstre' mod 'højre'

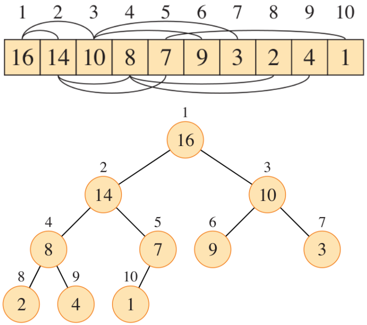


# Heaps

## En lidt mindre basal datastruktur

Nu hvor vi har set en række meget simple datastrukturer kaster vi os over en marginalt mere avanceret størrelse: et (binært) heap.

- Vi repræsenterer et heap med plads til $n$ elementer med et array $A[1:n]$, men det egentlige antal elementer angives med attributten $A.\mathit{heap\text{-}size}$
- Et heap kan ses som et næsten komplet træ, hvor alle niveauer er fyldt ud, pånær måske det sidste som dog skal være fyldt ud fra 'venstre' mod 'højre'
- Vi siger, at et element på plads $A[i]$ er parent node ('forældreknude') for elementerne på plads $A[2i]$ og $A[(2i+1]$, som dermed er venstre og højre barn

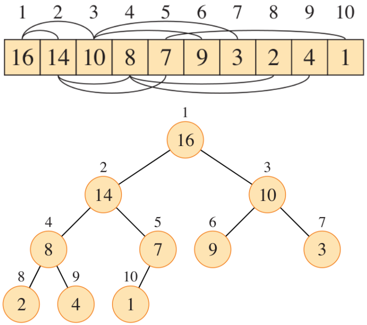


# Heaps

## En lidt mindre basal datastruktur

Nu hvor vi har set en række meget simple datastrukturer kaster vi os over en marginalt mere avanceret størrelse: et (binært) heap.

- Vi repræsenterer et heap med plads til $n$ elementer med et array $A[1:n]$, men det egentlige antal elementer angives med attributten $A.\mathit{heap\text{-}size}$
- Et heap kan ses som et næsten komplet træ, hvor alle niveauer er fyldt ud, pånær måske det sidste som dog skal være fyldt ud fra 'venstre' mod 'højre'
- Vi siger, at et element på plads $A[i]$ er parent node ('forældreknude') for elementerne på plads $A[2i]$ og $A[(2i+1]$, som dermed er venstre og højre barn
- En knude har en 'højde' (height) svarende til antal kanter på den længste vej fra roden til knuden. Højden på hele heapet er højden på roden, og da et heap er et (næsten) komplet binært træ er dets højde...

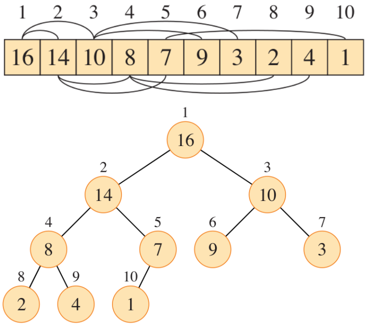


# Heaps

## En lidt mindre basal datastruktur

Nu hvor vi har set en række meget simple datastrukturer kaster vi os over en marginalt mere avanceret størrelse: et (binært) heap.

- Vi repræsenterer et heap med plads til $n$ elementer med et array $A[1:n]$, men det egentlige antal elementer angives med attributten $A.\mathit{heap\text{-}size}$
- Et heap kan ses som et næsten komplet træ, hvor alle niveauer er fyldt ud, pånær måske det sidste som dog skal være fyldt ud fra 'venstre' mod 'højre'
- Vi siger, at et element på plads $A[i]$ er parent node ('forældreknude') for elementerne på plads $A[2i]$ og $A[(2i+1]$, som dermed er venstre og højre barn
- En knude har en 'højde' (height) svarende til antal kanter på den længste vej fra roden til knuden. Højden på hele heapet er højden på roden, og da et heap er et (næsten) komplet binært træ er dets højde... $\Theta(\log n)$



# Heaps

## En lidt mindre basal datastruktur

Nu hvor vi har set en række meget simple datastrukturer kaster vi os over en marginalt mere avanceret størrelse: et (binært) heap.

Parent($i$)

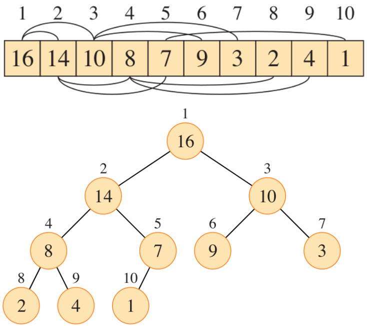
1 **return** $\lfloor i/2 \rfloor$

Left($i$)

1 **return** $2i$

Right($i$)

1 **return** $2i + 1$



Der er to forskellige slags heaps, som vi kalder hhv. Max-Heaps og Min-Heaps. Hver slags skal opfylde deres respektive heap-property:

### Max-Heap property

For alle knuder $i > 1$ gælder $A[\text{Parent}(i)] \geq A[i]$

Der er to forskellige slags heaps, som vi kalder hhv. Max-Heaps og Min-Heaps. Hver slags skal opfylde deres respektive heap-property:

**Max-Heap property**

For alle knuder $i > 1$ gælder $A[\text{Parent}(i)] \geq A[i]$

**Max-Heap property**

For alle knuder $i > 1$ gælder $A[\text{Parent}(i)] \leq A[i]$

Der er to forskellige slags heaps, som vi kalder hhv. Max-Heaps og Min-Heaps. Hver slags skal opfylde deres respektive heap-property:

**Max-Heap property**

For alle knuder $i > 1$ gælder $A[\text{Parent}(i)] \geq A[i]$

**Max-Heap property**

For alle knuder $i > 1$ gælder $A[\text{Parent}(i)] \leq A[i]$

Forskellen er minimial, så for nemheds skyld arbejder vi eksklusivt med Max-Heaps.

# Outline


1 Elementære datastrukturer

2 Heaps

3 Exercises

4 Operationer på heaps

5 Heapsort

6 Priority Queues

# Outline


1 Elementære datastrukturer

2 Heaps

3 Exercises

4 Operationer på heaps

5 Heapsort

6 Priority Queues

# Operationer på heaps

## Max-Heapify og Build-Max-Heap

Der er to grundlæggende operationer, der knytter sig til heaps:

# Operationer på heaps

## Max-Heapify og Build-Max-Heap

Der er to grundlæggende operationer, der knytter sig til heaps:

- Max-Heapify$(A, i)$ sikrer, at (max-)heap-egenskaben opretholdes i træet med rod i $A[i]$

# Operationer på heaps

## Max-Heapify og Build-Max-Heap

Der er to grundlæggende operationer, der knytter sig til heaps:

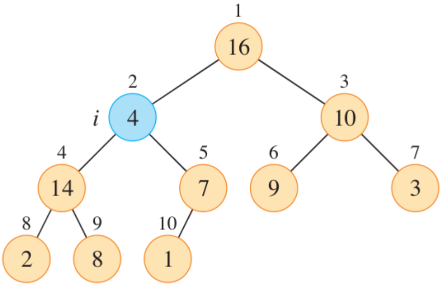
- Max-Heapify($A, i$) sikrer, at (max-)heap-egenskaben opretholdes i træet med rod i $A[i]$
- Build-Max-Heap($A$) tager et arbitrært array $A$ og konverterer det til et (max-)heap

# Operationer på heaps

## Max-Heapify

Vi starter med at se på Max-Heapify.

- Formålet er at opretholde max-heap egenskaben
- Inputtet er et array $A$ og et index $i$, hvor Left($i$) og Right($i$) er max-heaps, men hvor $A[i]$ måske er mindre end Left($i$) og Right($i$)
- Outputtet er det omorganiserede array, der nu overholder max-heap egenskaben i $A[i]$

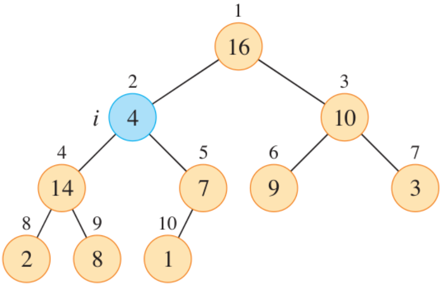


# Operationer på heaps

## Max-Heapify

Vi starter med at se på Max-Heapify.

- Vi starter med at sammenligne $A[i]$ med $A[\text{Left}(i)]$ og $A[\text{Right}(i)]$ for at finde ud af, hvilken der er størst

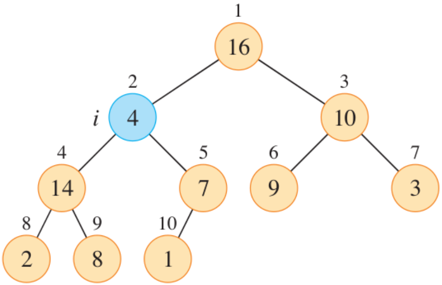


# Operationer på heaps

## Max-Heapify

Vi starter med at se på Max-Heapify.

- Vi starter med at sammenligne $A[i]$ med $A[\text{Left}(i)]$ og $A[\text{Right}(i)]$ for at finde ud af, hvilken der er størst
- Så lader vi $A[i]$ 'synke' ned i træet ved at bytte det ud med det største af sine børn

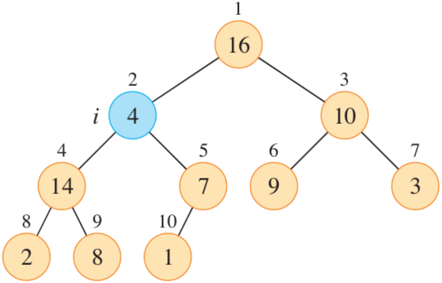


# Operationer på heaps

## Max-Heapify

Vi starter med at se på Max-Heapify.

- Vi starter med at sammenligne $A[i]$ med $A[\text{Left}(i)]$ og $A[\text{Right}(i)]$ for at finde ud af, hvilken der er størst
- Så lader vi $A[i]$ 'synke' ned i træet ved at bytte det ud med det største af sine børn
- Vi fortsætter rekursivt til $A[i]$ enten er størst eller et blad

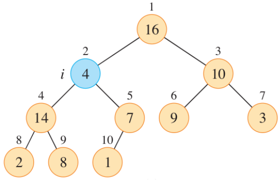


# Max-Heapify

## Pseudo-kode

**Max-Heapify($A$, $i$)**

```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heapify(A, largest)
```

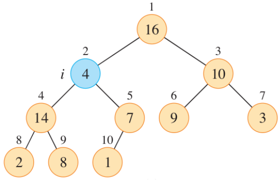


# Max-Heapify

## Pseudo-kode

**Max-Heapify($A$, $i$)**

```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heapify(A, largest)
```



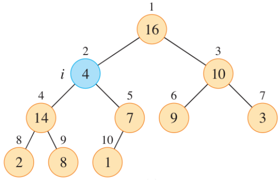

- Tjek om venstre barn er i heapet og om det er større end $A[i]$

# Max-Heapify

## Pseudo-kode

### Max-Heapify($A$, $i$)

```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heapify(A, largest)
```



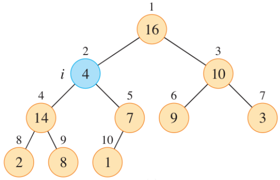

- Tjek om venstre barn er i heapet og om det er større end $A[i]$
- Gem hvadend der er størst af $A[i]$ og $A[\text{Left}(i)]$ i variablen *largest*

# Max-Heapify

## Pseudo-kode

**Max-Heapify($A$, $i$)**

```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heapify(A, largest)
```



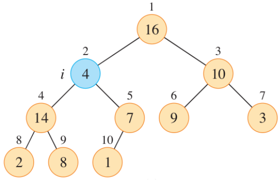

- Tjek om venstre barn er i heapet og om det er større end $A[i]$
- Gem hvadend der er størst af $A[i]$ og $A[\text{Left}(i)]$ i variablen *largest*
- Tjek om højre barn er i heapet og om det er større end *largest* (og opdater evt *largest*)

# Max-Heapify

## Pseudo-kode

**Max-Heapify($A$, $i$)**

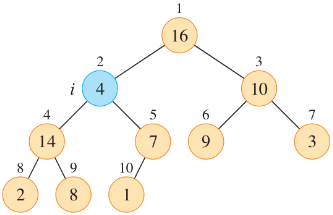
```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heapify(A, largest)
```




- Tjek om venstre barn er i heapet og om det er større end $A[i]$
- Gem hvadend der er størst af $A[i]$ og $A[\text{Left}(i)]$ i variablen *largest*
- Tjek om højre barn er i heapet og om det er større end *largest* (og opdater evt *largest*)
- Hvis det ikke er $A[i]$, der er størst, byt $A[i]$ ud med $A[\textit{largest}]$ og kald rekursivt

# Max-Heapify

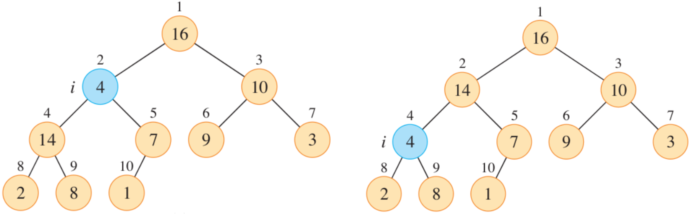
## Eksempel




- Vi har kaldt Max-Heapify$(A, 2)$

# Max-Heapify

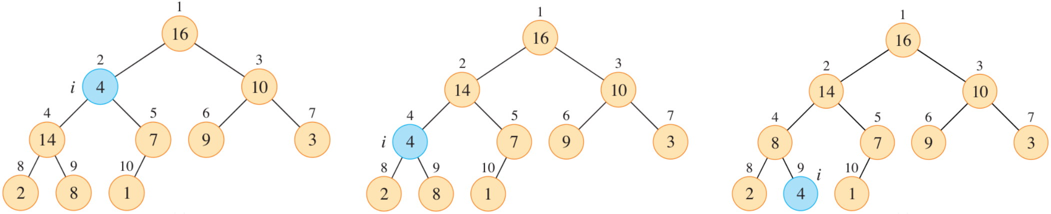
## Eksempel




- Vi har kaldt Max-Heapify$(A, 2)$
- $A[2] = 4$ hvilket er mindre end begge børn, så vi bytter med det største barn, $A[4]$, og kalder Max-Heapify$(A, 4)$

# Max-Heapify

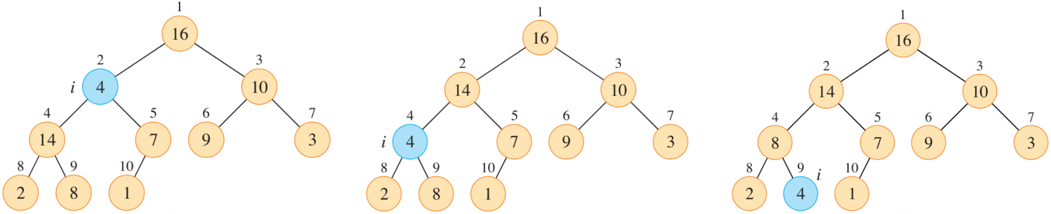
## Eksempel




- Vi har kaldt Max-Heapify$(A, 2)$
- $A[2] = 4$ hvilket er mindre end begge børn, så vi bytter med det største barn, $A[4]$, og kalder Max-Heapify$(A, 4)$
- Nu har vi $A[4] = 4$, hvilket er større end det venstre barn men mindre end det højre. Vi bytter dem og kalder Max-Heapify$(A, 9)$

# Max-Heapify

## Eksempel




- Vi har kaldt Max-Heapify$(A, 2)$
- $A[2] = 4$ hvilket er mindre end begge børn, så vi bytter med det største barn, $A[4]$, og kalder Max-Heapify$(A, 4)$
- Nu har vi $A[4] = 4$, hvilket er større end det venstre barn men mindre end det højre. Vi bytter dem og kalder Max-Heapify$(A, 9)$
- $A[9]$ har ingen børn, så rekursionen slutter nu, og max-heap egenskaben er overholdt

# Max-Heapify

## Tidsanalyse

- Linie 1-9 er...

**Max-Heapify($A, i$)**

```
1   l = Left(i)
2   r = Right(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4           largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7           largest = r
8   if largest ≠ i
9           exchange A[i] with A[largest]
10          Max-Heapify(A, largest)
```

# Max-Heapify

## Tidsanalyse

- Linie 1-9 er... $\Theta(1)$

**Max-Heapify($A, i$)**

```
1   l = Left(i)
2   r = Right(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4         largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7         largest = r
8   if largest ≠ i
9         exchange A[i] with A[largest]
10        Max-Heapify(A, largest)
```

# Max-Heapify

## Tidsanalyse

AALBORG UNIVERSITET

- Linie 1-9 er... $\Theta(1)$
- Hvad med linie 10?

**Max-Heapify$(A, i)$**

```
1   l = Left(i)
2   r = Right(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4         largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7         largest = r
8   if largest ≠ i
9         exchange A[i] with A[largest]
10        Max-Heapify(A, largest)
```

# Max-Heapify

## Tidsanalyse

- Linie 1-9 er... $\Theta(1)$
- Hvad med linie 10?
  - Worst case er, at nederste niveau i træet er halvt fuldt (hvorfor?) — i så fald har det største sub-træ maks $2n/3$ knuder

**Max-Heapify($A$, $i$)**

```
1   l = Left(i)
2   r = Right(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4           largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7           largest = r
8   if largest ≠ i
9           exchange A[i] with A[largest]
10          Max-Heapify(A, largest)
```

# Max-Heapify

## Tidsanalyse



- Linie 1-9 er... $\Theta(1)$
- Hvad med linie 10?
  - Worst case er, at nederste niveau i træet er halvt fuldt (hvorfor?) — i så fald har det største sub-træ maks $2n/3$ knuder
  - Dermed er størrelsen af sub-problemet i worst-case $T(2n/3)$

**Max-Heapify($A$, $i$)**

```
1   l = Left(i)
2   r = Right(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4           largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7           largest = r
8   if largest ≠ i
9           exchange A[i] with A[largest]
10          Max-Heapify(A, largest)
```

# Max-Heapify

## Tidsanalyse

- Linie 1-9 er… $\Theta(1)$
- Hvad med linie 10?
  - Worst case er, at nederste niveau i træet er halvt fuldt (hvorfor?) — i så fald har det største sub-træ maks $2n/3$ knuder
  - Dermed er størrelsen af sub-problemet i worst-case $T(2n/3)$
- Vi får altså en rekursion på formen $T(n) = T(2n/3) + \Theta(1)$

**Max-Heapify$(A, i)$**

1 $l$ = Left$(i)$
2 $r$ = Right$(i)$
3 **if** $l \leq A.\mathit{heap\text{-}size}$ and $A[l] > A[i]$
4 $\quad$ $\mathit{largest} = l$
5 **else** $\mathit{largest} = i$
6 **if** $r \leq A.\mathit{heap\text{-}size}$ and $A[r] > A[\mathit{largest}]$
7 $\quad$ $\mathit{largest} = r$
8 **if** $\mathit{largest} \neq i$
9 $\quad$ exchange $A[i]$ with $A[\mathit{largest}]$
10 $\quad$ Max-Heapify$(A, \mathit{largest})$

# Bonus-analyse: Træstruktur

## Worst case størrelse af barn

- Først, bemærk at det værste split vil være, hvis det ene sub-træ er komplet med højde $h$ og det andet komplet med højde $h - 1$

# Bonus-analyse: Træstruktur

## Worst case størrelse af barn

- Først, bemærk at det værste split vil være, hvis det ene sub-træ er komplet med højde $h$ og det andet komplet med højde $h - 1$
- Et komplet binært træ med $n$ knuder vil have $k$ 'interne knuder' og præcis $k + 1$ blade (hvorfor?) — altså er $n = 2k + 1$

# Bonus-analyse: Træstruktur

## Worst case størrelse af barn

- Først, bemærk at det værste split vil være, hvis det ene sub-træ er komplet med højde $h$ og det andet komplet med højde $h-1$
- Et komplet binært træ med $n$ knuder vil have $k$ 'interne knuder' og præcis $k+1$ blade (hvorfor?) — altså er $n = 2k+1$
- Lad os sige, at det store sub-træ har $2k+1$ knuder, mens det lille, der mangler det sidste niveau af $k+1$ blade, kun har $k$ knuder

# Bonus-analyse: Træstruktur

## Worst case størrelse af barn

- Først, bemærk at det værste split vil være, hvis det ene sub-træ er komplet med højde $h$ og det andet komplet med højde $h-1$
- Et komplet binært træ med $n$ knuder vil have $k$ 'interne knuder' og præcis $k+1$ blade (hvorfor?) — altså er $n = 2k+1$
- Lad os sige, at det store sub-træ har $2k+1$ knuder, mens det lille, der mangler det sidste niveau af $k+1$ blade, kun har $k$ knuder
- Forældreknuden må således have $n = 1 + (2k+1) + k = 3k+2$ knuder

# Bonus-analyse: Træstruktur

## Worst case størrelse af barn

- Først, bemærk at det værste split vil være, hvis det ene sub-træ er komplet med højde $h$ og det andet komplet med højde $h - 1$
- Et komplet binært træ med $n$ knuder vil have $k$ 'interne knuder' og præcis $k + 1$ blade (hvorfor?) — altså er $n = 2k + 1$
- Lad os sige, at det store sub-træ har $2k + 1$ knuder, mens det lille, der mangler det sidste niveau af $k + 1$ blade, kun har $k$ knuder
- Forældreknuden må således have $n = 1 + (2k + 1) + k = 3k + 2$ knuder
- Det kan vi omskrive, så vi på højresiden får størrelsen af det store sub-træ:

$$
\begin{aligned}
& n = 3k + 2 \\
\implies & \frac{n}{3} = k + \frac{2}{3} \\
\implies & \frac{2n}{3} = 2k + \frac{4}{3} \\
\implies & \frac{2n}{3} - \frac{1}{3} = 2k + 1
\end{aligned}
$$

# Bonus-analyse: Træstruktur

## Worst case størrelse af barn

- Først, bemærk at det værste split vil være, hvis det ene sub-træ er komplet med højde $h$ og det andet komplet med højde $h - 1$
- Et komplet binært træ med $n$ knuder vil have $k$ 'interne knuder' og præcis $k + 1$ blade (hvorfor?) — altså er $n = 2k + 1$
- Lad os sige, at det store sub-træ har $2k + 1$ knuder, mens det lille, der mangler det sidste niveau af $k + 1$ blade, kun har $k$ knuder
- Forældreknuden må således have $n = 1 + (2k + 1) + k = 3k + 2$ knuder
- Det kan vi omskrive, så vi på højresiden får størrelsen af det store sub-træ:

$$
\begin{aligned}
& n = 3k + 2 \\
\implies & \frac{n}{3} = k + \frac{2}{3} \\
\implies & \frac{2n}{3} = 2k + \frac{4}{3} \\
\implies & \frac{2n}{3} - \frac{1}{3} = 2k + 1
\end{aligned}
$$

- Altså er $\frac{2n}{3}$ et upper bound på antal knuder i et subtræ til et træ med størrelse $n$

# Max-Heapify

## Tidsanalyse

Så altså… Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så?

# Max-Heapify

## Tidsanalyse

Så altså… Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

# Max-Heapify

## Tidsanalyse

Så altså... Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a =$ , $b =$ og $f(n) =$

# Max-Heapify
## Tidsanalyse

Så altså... Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a = 3/2$, $b =$ og $f(n) =$

# Max-Heapify
## Tidsanalyse

Så altså... Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a = 3/2$, $b = 1$ og $f(n) =$

# Max-Heapify

## Tidsanalyse

Så altså... Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a = 3/2$, $b = 1$ og $f(n) = \Theta(1)$

Så altså... Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a = 3/2$, $b = 1$ og $f(n) = \Theta(1)$
- Vi indsætter og forenkler: $n^{\log_{3/2} 1} = n^0 = 1$

# Max-Heapify

## Tidsanalyse

Så altså. . . Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a = 3/2$, $b = 1$ og $f(n) = \Theta(1)$
- Vi indsætter og forenkler: $n^{\log_{3/2} 1} = n^0 = 1$
- Vi sammenligner med $f(n)$:

$$f(n) = \begin{matrix} O(n^{0-\epsilon}) \\ \Theta(n^0 \log^k n) \\ \Omega(n^{0+\epsilon}) \end{matrix}$$

# Max-Heapify

## Tidsanalyse

Så altså... Sub-problemet i det rekursive kald i Max-Hepify har i værste fald en størrelse på $2n/3$ mens resten er konstant. Kan vi bruge master method så? Ja!

- Vi har $T(n) = T(2n/3) + \Theta(1)$
- Dermed: $a = 3/2$, $b = 1$ og $f(n) = \Theta(1)$
- Vi indsætter og forenkler: $n^{\log_{3/2} 1} = n^0 = 1$
- Vi sammenligner med $f(n)$:

$$f(n) = \begin{cases} O(n^{0-\epsilon}) \\ \Theta(n^0 \log^k n) \\ \Omega(n^{0+\epsilon}) \end{cases}$$

- Dette er case 2 i teoremet, når vi vælger $k = 0$, hvilket giver os en køretid $T(n) = \Theta(n^0 \log^{k+1} n) = \Theta(\log n)$

# Build-Max-Heap

## Nu bygger vi et heap!

Den næste operation, vi ser på, har det simple formål at konstruere et heap ud fra et uorganiseret array $A$.

- Bemærk først, at den sidste halvdel af $A$ kan betragtes som blade i træet og er dermed trivielt korrekte max-heaps (ie. overholder heap-egenskaben)

**Build-Max-Heap($A$, $n$)**

1 $A.\mathit{heap\text{-}size} = n$
2 **for** $i = \lfloor n/2 \rfloor$ **downto** 1
3 Max-Heapify($A$, $i$)

# Build-Max-Heap

## Nu bygger vi et heap!

Den næste operation, vi ser på, har det simple formål at konstruere et heap ud fra et uorganiseret array $A$.

- Bemærk først, at den sidste halvdel af $A$ kan betragtes som blade i træet og er dermed trivielt korrekte max-heaps (ie. overholder heap-egenskaben)
- Dermed skal vi bare kalde Max-Heapify på elementerne fra plads $n/2$ og ned til 1 for at have håndhævet heap-egenskaben på hele arrayet

**Build-Max-Heap($A$, $n$)**

```
A.heap-size = n
for i = ⌊n/2⌋ downto 1
        Max-Heapify(A, i)
```

# Build-Max-Heap

## Nu bygger vi et heap!

Den næste operation, vi ser på, har det simple formål at konstruere et heap ud fra et uorganiseret array $A$.

- Bemærk først, at den sidste halvdel af $A$ kan betragtes som blade i træet og er dermed trivielt korrekte max-heaps (ie. overholder heap-egenskaben)
- Dermed skal vi bare kalde Max-Heapify på elementerne fra plads $n/2$ og ned til 1 for at have håndhævet heap-egenskaben på hele arrayet
  - Husk at Max-Heapify kaldt på index $i$ forventer, at Left($i$) og Right($i$) overholder heap-egenskaben

**Build-Max-Heap($A$, $n$)**

```
1   A.heap-size = n
2   for i = ⌊n/2⌋ downto 1
3         Max-Heapify(A, i)
```

# Build-Max-Heap

## Nu bygger vi et heap!

Den næste operation, vi ser på, har det simple formål at konstruere et heap ud fra et uorganiseret array $A$.

- Bemærk først, at den sidste halvdel af $A$ kan betragtes som blade i træet og er dermed trivielt korrekte max-heaps (ie. overholder heap-egenskaben)
- Dermed skal vi bare kalde Max-Heapify på elementerne fra plads $n/2$ og ned til 1 for at have håndhævet heap-egenskaben på hele arrayet
    - Husk at Max-Heapify kaldt på index $i$ forventer, at Left($i$) og Right($i$) overholder heap-egenskaben
- Køretiden for linie 2 er $O(n)$ og vi har lige vist at Max-Hepify er i $\Theta(\log n)$ — altså en samlet køretid på $O(n \log n)$

**Build-Max-Heap($A, n$)**

```
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3          Max-Heapify(A, i)
```

# Build-Max-Heap

## Nu bygger vi et heap!

Den næste operation, vi ser på, har det simple formål at konstruere et heap ud fra et uorganiseret array $A$.

- Bemærk først, at den sidste halvdel af $A$ kan betragtes som blade i træet og er dermed trivielt korrekte max-heaps (ie. overholder heap-egenskaben)
- Dermed skal vi bare kalde Max-Heapify på elementerne fra plads $n/2$ og ned til 1 for at have håndhævet heap-egenskaben på hele arrayet
  - Husk at Max-Heapify kaldt på index $i$ forventer, at Left($i$) og Right($i$) overholder heap-egenskaben
- Køretiden for linie 2 er $O(n)$ og vi har lige vist at Max-Hepify er i $\Theta(\log n)$ — altså en samlet køretid på $O(n \log n)$
  - Dog...Med lidt snilde kan man faktisk udlede et strammere bound på $\Theta(n)$ — se CLRS 6.3

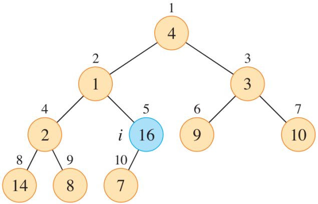
**Build-Max-Heap($A$, $n$)**

```
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3          Max-Heapify(A, i)
```

# Build-Max-Heap

## Eksempel

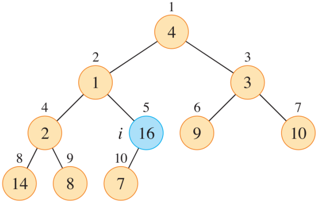
Vi kalder proceduren på inputtet $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.

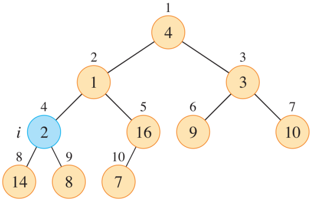


# Build-Max-Heap

## Eksempel

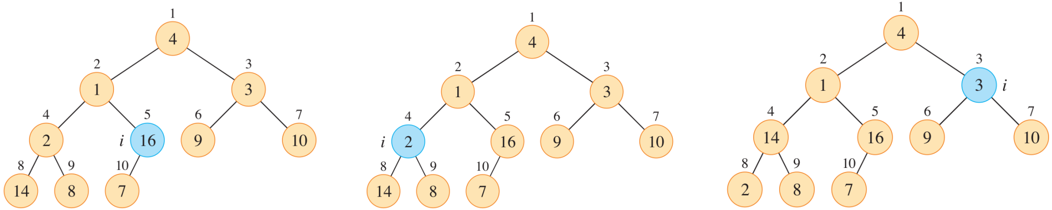
Vi kalder proceduren på inputtet $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.



# Build-Max-Heap
## Eksempel

Vi kalder proceduren på inputtet $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.

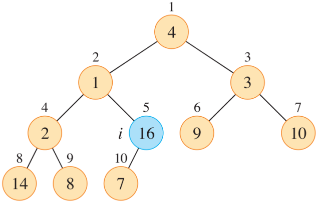


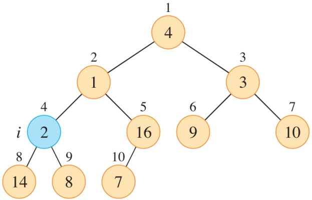
# Build-Max-Heap

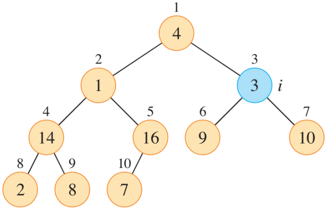
## Eksempel

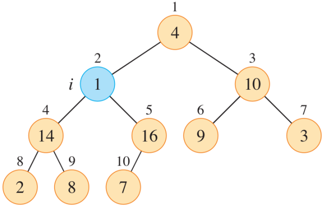


Vi kalder proceduren på inputtet $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.



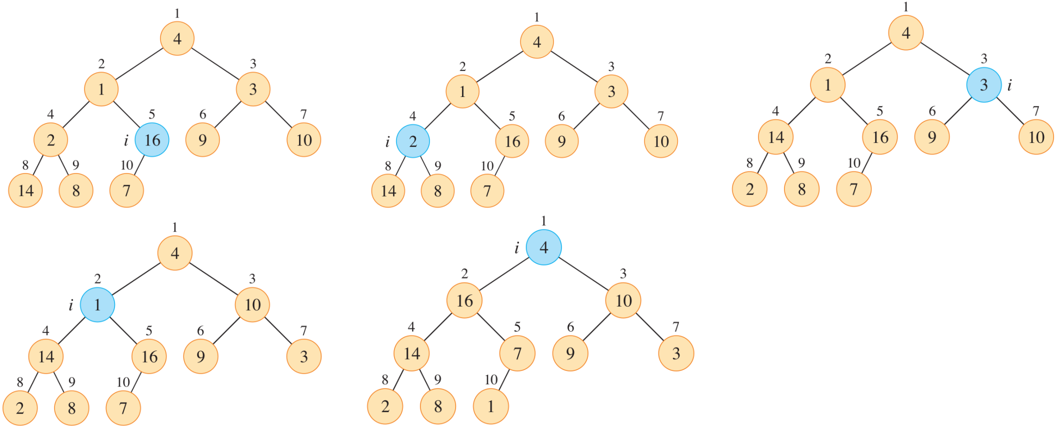



# Build-Max-Heap

## Eksempel

Vi kalder proceduren på inputtet $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.

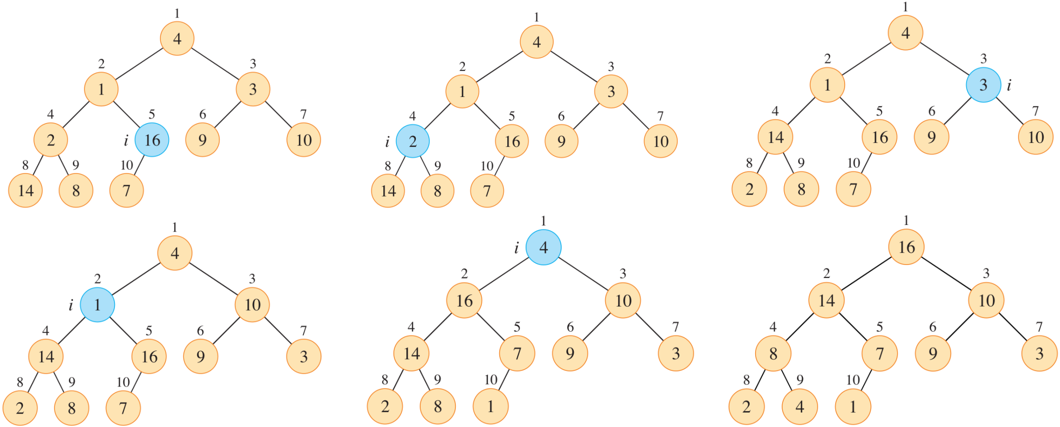


# Build-Max-Heap

## Eksempel



Vi kalder proceduren på inputtet $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.



# Outline

1 Elementære datastrukturer

2 Heaps

3 Exercises

4 Operationer på heaps

5 Heapsort

6 Priority Queues

# Heapsort

## Sort of simpelt

Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

# Heapsort

## Sort of simpelt

Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

- Vi skal sortere $A$, så vi starter med at konvertere det til et max-heap

**Heapsort($A$, $n$)**

Build-Max-Heap($A$, $n$)
**for** $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    Max-Heapify($A$, 1)

# Heapsort

## Sort of simpelt

AALBORG UNIVERSITET

Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

- Vi skal sortere $A$, så vi starter med at konvertere det til et max-heap
- Nu ved vi, at første element er det største — altså skal det stå sidst i det sorterede array

**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

# Heapsort

## Sort of simpelt

Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

- Vi skal sortere $A$, så vi starter med at konvertere det til et max-heap
- Nu ved vi, at første element er det største — altså skal det stå sidst i det sorterede array
- Vi bytter derfor det første og sidste element (som er et blad) med hinanden og gør $A.\mathit{heap\text{-}size}$ en mindre

**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

# Heapsort

## Sort of simpelt



Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

- Vi skal sortere $A$, så vi starter med at konvertere det til et max-heap
- Nu ved vi, at første element er det største — altså skal det stå sidst i det sorterede array
- Vi bytter derfor det første og sidste element (som er et blad) med hinanden og gør $A.\mathit{heap\text{-}size}$ en mindre
- Så kalder vi Max-Heapify på det nye forreste element, og genopretter dermed heap-egenskaben

**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

# Heapsort

## Sort of simpelt

Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

- Vi skal sortere $A$, så vi starter med at konvertere det til et max-heap
- Nu ved vi, at første element er det største — altså skal det stå sidst i det sorterede array
- Vi bytter derfor det første og sidste element (som er et blad) med hinanden og gør $A.heap\text{-}size$ en mindre
- Så kalder vi Max-Heapify på det nye forreste element, og genopretter dermed heap-egenskaben
- Dette fortsætter vi med, indtil vi har været hele arrayet igennem — til sidst er heapet tomt men arrayet sorteret i stigende rækkefølge!

**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        Max-Heapify(A, 1)
```

# Heapsort

## Sort of simpelt

Nu da vi har Max-Heapify og Build-Max-Heap, som er to hurtige operationer, kan I så forestille jer, hvordan vi kan kombinere de to til en sorteringsalgoritme?

- Vi skal sortere $A$, så vi starter med at konvertere det til et max-heap
- Nu ved vi, at første element er det største — altså skal det stå sidst i det sorterede array
- Vi bytter derfor det første og sidste element (som er et blad) med hinanden og gør $A.\mathit{heap\text{-}size}$ en mindre
- Så kalder vi Max-Heapify på det nye forreste element, og genopretter dermed heap-egenskaben
- Dette fortsætter vi med, indtil vi har været hele arrayet igennem — til sidst er heapet tomt men arrayet sorteret i stigende rækkefølge!
- Kompleksitet?

**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

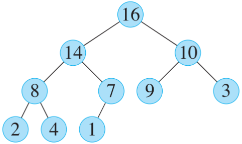
# Heapsort

## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:

### Heapsort$(A, n)$

Build-Max-Heap$(A, n)$
**for** $i = n$ **downto** 2
    swap $A[1]$ and $A[i]$
    $A.\mathit{heap\text{-}size} --$
    Max-Heapify$(A, 1)$

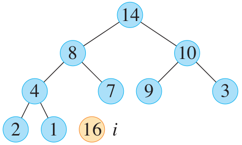



Current status $A = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

# Heapsort

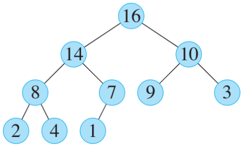
## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:




### Heapsort($A$, $n$)

Build-Max-Heap($A$, $n$)
**for** $i = n$ **downto** 2
    swap $A[1]$ and $A[i]$
    $A.heap\text{-}size - -$
    Max-Heapify($A$, 1)



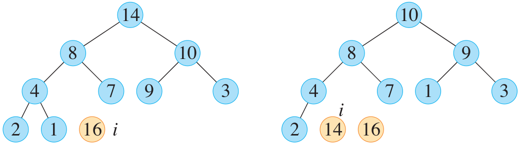

Current status $A = \langle 14, 8, 10, 4, 7, 9, 3, 2, 1, 16 \rangle$

# Heapsort

## Sort of simpelt

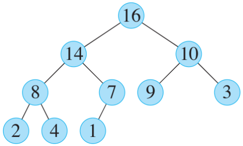


Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:




**Heapsort$(A, n)$**

```
Build-Max-Heap(A, n)
for i = n downto 2
    swap A[1] and A[i]
    A.heap-size − −
    Max-Heapify(A, 1)
```

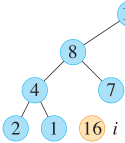


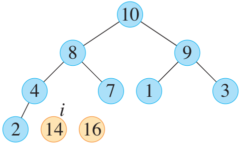

Current status $A = \langle 10, 8, 9, 4, 7, 1, 3, 2, 14, 16 \rangle$

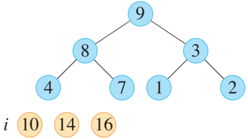
# Heapsort

## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:

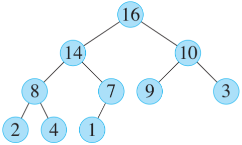



### Heapsort$(A, n)$

Build-Max-Heap$(A, n)$
**for** $i = n$ **downto** 2
  swap $A[1]$ and $A[i]$
  $A.heap\text{-}size$ − −
  Max-Heapify$(A, 1)$

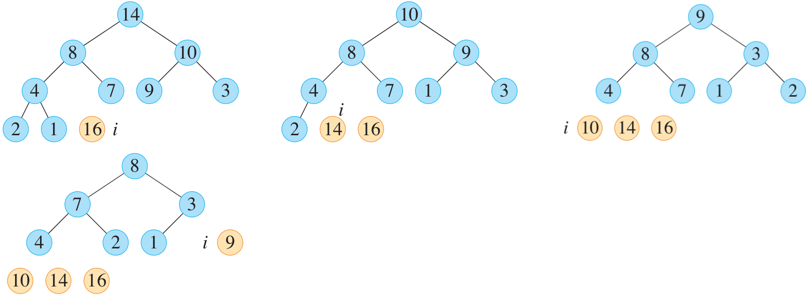



Current status $A = \langle 9, 8, 3, 4, 7, 1, 2, 10, 14, 16 \rangle$

# Heapsort

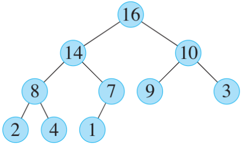
## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:



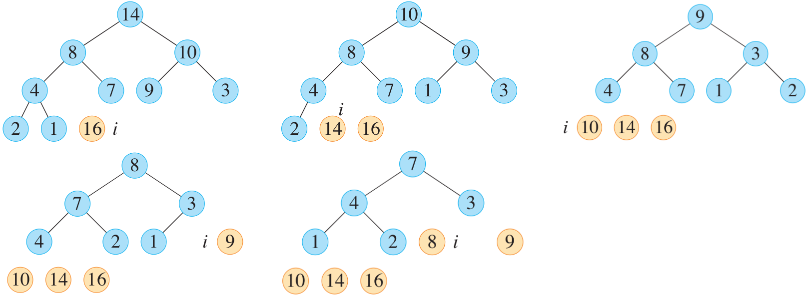

### Heapsort$(A, n)$

Build-Max-Heap$(A, n)$
**for** $i = n$ **downto** 2
  swap $A[1]$ and $A[i]$
  $A.\,heap\text{-}size - -$
  Max-Heapify$(A, 1)$



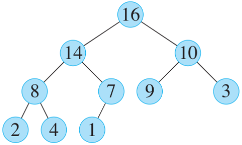

Current status $A = \langle 8, 7, 3, 4, 2, 1, 9, 10, 14, 16 \rangle$

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:



**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    swap A[1] and A[i]
    A.heap-size − −
    Max-Heapify(A, 1)
```

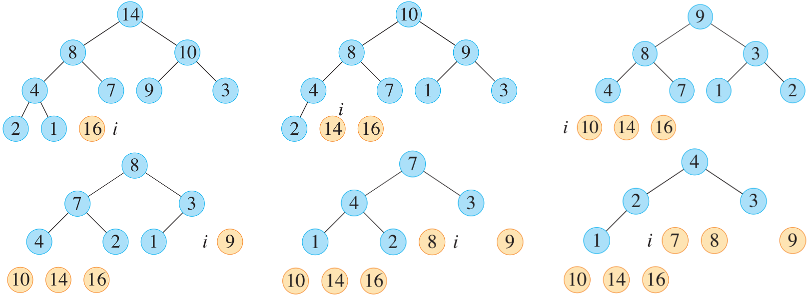


Current status $A = \langle 7, 4, 3, 1, 2, 8, 9, 10, 14, 16 \rangle$

# Heapsort

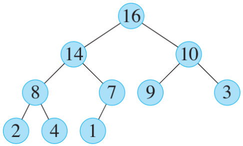
## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:



**Heapsort($A$, $n$)**

```
Build-Max-Heap(A, n)
for i = n downto 2
    swap A[1] and A[i]
    A.heap-size − −
    Max-Heapify(A, 1)
```

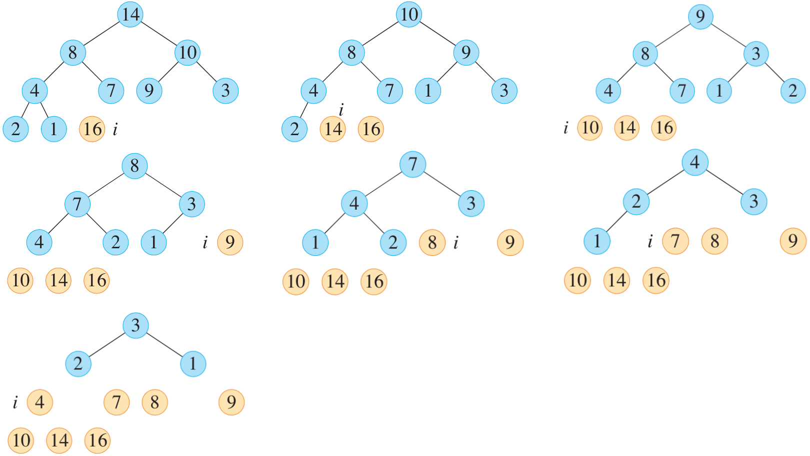


Current status $A = \langle 4, 2, 3, 1, 7, 8, 9, 10, 14, 16 \rangle$

# Heapsort

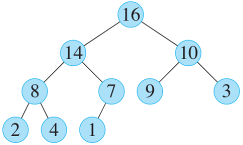
## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:



**Heapsort($A$, $n$)**

1. Build-Max-Heap($A$, $n$)
2. **for** $i = n$ **downto** 2
3. swap $A[1]$ and $A[i]$
4. $A.heap\text{-}size --$
5. Max-Heapify($A$, 1)

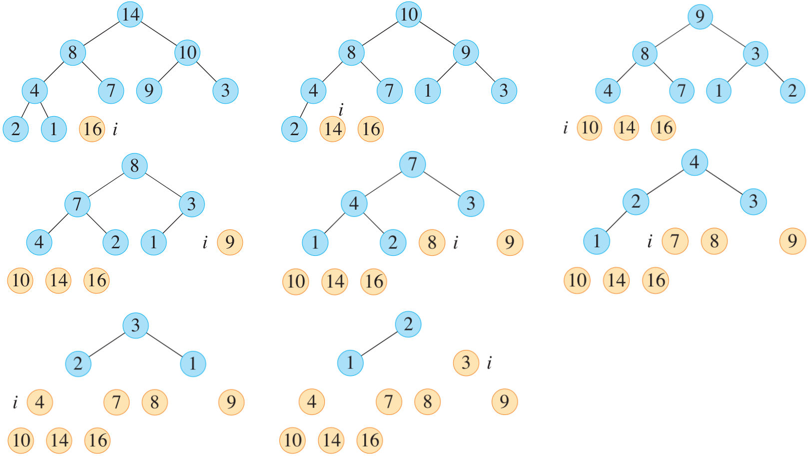


Current status $A = \langle 3, 2, 1, 4, 7, 8, 9, 10, 14, 16 \rangle$

# Heapsort

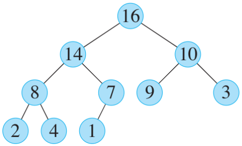
## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:



### Heapsort($A$, $n$)

```
Build-Max-Heap(A, n)
for i = n downto 2
    swap A[1] and A[i]
    A.heap-size − −
    Max-Heapify(A, 1)
```

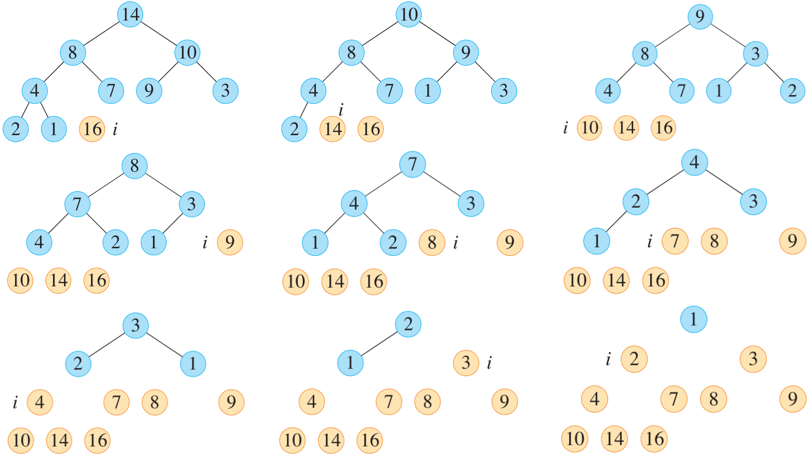


Current status $A = \langle 2, 1, 3, 4, 7, 8, 9, 10, 14, 16 \rangle$

# Heapsort

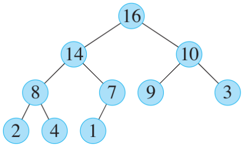
## Sort of simpelt

Vi vil sortere sekvensen $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$:



**Heapsort$(A, n)$**

```
Build-Max-Heap(A, n)
for i = n downto 2
    swap A[1] and A[i]
    A.heap-size − −
    Max-Heapify(A, 1)
```



Current status $A = \langle 1, 2, 3, 4, 7, 8, 9, 10, 14, 16 \rangle$

# Outline


1. Elementære datastrukturer
2. Heaps
3. Exercises
4. Operationer på heaps
5. Heapsort
6. Priority Queues

# Priority queues
## Prioriterede køer

Heapsort er god, fordi den både kører i $\Theta(n \log n)$ tid og sorter in-place, og dermed kun kræver $\Theta(1)$ ekstra plads. Men i praksis er quicksort typisk hurtigere... Heaps hart dog et andet trick i ærmet, nemlig priority queues!

- Abstrakt datastruktur, der understøtter operationerne Insert, Extract-Maximum og Maximum
- En dynamisk kø, hvor rækkefølgen ikke er bestemt af indsættelsesrækkefølgen men af en given key
- Forestil jer en skadestue, hvor ham med den forstuvede finger måske dukkede op først, men hende med en punkteret lunge alligevel skal forrest i køen (eller I kan forestille jer noget mindre makabert, såsom en job scheduler i en computer eller noget andet kedeligt)

# Priority queues

## Operationer

Vi kigger først på Insert-proceduren.

- Vi tager som input et heap $A$ og et element $x$, vi vil indsætte

**Insert($A$, $x$)**

$A.\mathit{heap\text{-}size} = A.\mathit{heap\text{-}size} + 1$
$A[A.\mathit{heap\text{-}size}] = x$
$i = A.\mathit{heap\text{-}size}$
**while** $i > 1$ and $A[\mathrm{Parent}(i).\mathit{key}] < A[i].\mathit{key}$
    exchange $A[i]$ with $A[\mathrm{Parent}(i)]$
    $i = \mathrm{Parent}(i)$

# Priority queues

## Operationer



Vi kigger først på Insert-proceduren.

- Vi tager som input et heap $A$ og et element $x$, vi vil indsætte
- Vi øger $A.\mathit{heap\text{-}size}$ med 1 for at gøre plads (tjekker selvfølgelig, at $A.\mathit{heap\text{-}size} < A.\mathit{length}$)

**Insert($A, x$)**

$A.\mathit{heap\text{-}size} = A.\mathit{heap\text{-}size} + 1$
$A[A.\mathit{heap\text{-}size}] = x$
$i = A.\mathit{heap\text{-}size}$
**while** $i > 1$ and $A[\text{Parent}(i).\mathit{key}] < A[i].\mathit{key}$
  exchange $A[i]$ with $A[\text{Parent}(i)]$
  $i = \text{Parent}(i)$

# Priority queues

## Operationer

Vi kigger først på Insert-proceduren.

- Vi tager som input et heap $A$ og et element $x$, vi vil indsætte
- Vi øger $A.\mathit{heap\text{-}size}$ med 1 for at gøre plads (tjekker selvfølgelig, at $A.\mathit{heap\text{-}size} < A.\mathit{length}$)
- Vi indsætter $x$ bagerst i heapet og lader det 'svømme' opad, ved at bytte det ud med sin forældrer, så længe det er større

**Insert$(A, x)$**

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = x
i = A.heap-size
while i > 1 and A[Parent(i).key] < A[i].key
    exchange A[i] with A[Parent(i)]
    i = Parent(i)
```

# Priority queues

## Operationer

Vi kigger først på Insert-proceduren.

- Vi tager som input et heap $A$ og et element $x$, vi vil indsætte
- Vi øger $A.\mathit{heap\text{-}size}$ med 1 for at gøre plads (tjekker selvfølgelig, at $A.\mathit{heap\text{-}size} < A.\mathit{length}$)
- Vi indsætter $x$ bagerst i heapet og lader det 'svømme' opad, ved at bytte det ud med sin forældrer, så længe det er større
- Kompleksitet?

### Insert($A$, $x$)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = x
i = A.heap-size
while i > 1 and A[Parent(i).key] < A[i].key
    exchange A[i] with A[Parent(i)]
    i = Parent(i)
```

Vi kigger først på Insert-proceduren.

- Vi tager som input et heap $A$ og et element $x$, vi vil indsætte
- Vi øger $A.\mathit{heap\text{-}size}$ med 1 for at gøre plads (tjekker selvfølgelig, at $A.\mathit{heap\text{-}size} < A.\mathit{length}$)
- Vi indsætter $x$ bagerst i heapet og lader det 'svømme' opad, ved at bytte det ud med sin forældrer, så længe det er større
- Kompleksitet?
  - Siden hver iteration af while-løkken flytter elementet et niveau op i træet, så udgør højden af træet et tight bound — $\Theta(\log n)$

**Insert($A, x$)**

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = x
i = A.heap-size
while i > 1 and A[Parent(i).key] < A[i].key
    exchange A[i] with A[Parent(i)]
    i = Parent(i)
```

# Priority queues

## The CLRS way

Bemærk, at CLRS gør det en smule anderledes. Det er essentielt set det samme, så det er fint, hvis I bare forstår Insert på sidste slide.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = −∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

# Priority queues

## Extract-Maximum

Sidst men ikke mindst, så ser vi på, hvordan vi kan slette det største element fra en priority queue.

- Vi gemmer det største element (som også er det første element) i en variable *max*
- Vi flytter det sidste element frem til at stå på index 1
- Vi sænker $A.\mathit{heap\text{-}size}$
- Vi kalder Max-Heapify$(A, 1)$ og reetablerer dermed heap-egenskaben
- Slutteligt returnerer vi *max*
- Og glædeligt nok er denne operation også $\Theta(\log n)$!

**Heap-Extract-Max$(A)$**

```
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
Max-Heapify(A, 1)
return max
```

# Slut på forelæsning 4

## Puh!

Endelig. . .

# Dagens temaer

## Opsummering

- Vi har mødt basale datastrukturer som stacks, queues og linked lists
  - Stacks følger LIFO ('last in, first out')
  - Queues følger FIFI ('first in, first out')
  - Linked lists erstatter array-fundamentet med pointers fra hvert element til det næste (og til det forrige, hvis det er en doubly linked list)

# Dagens temaer
## Opsummering

- Vi har mødt basale datastrukturer som stacks, queues og linked lists
  - Stacks følger LIFO ('last in, first out')
  - Queues følger FIFI ('first in, first out')
  - Linked lists erstatter array-fundamentet med pointers fra hvert element til det næste (og til det forrige, hvis det er en doubly linked list)
- Vi har mødt heaps og lært
  - at et heap, der kan fortolkes som et binært træ
  - at et max-heap skal overholde heap-egenskaben: 'For alle $i > 1$ gælder $A[\mathrm{Parent}(i)] \geq A[i]$'
  - at vi med Max-Heapify i $\Theta(\log n)$ tid kan re-etablere heap-egenskaben i en knude $A[i]$, så længe Left($i$) og Right($i$) er rødder i træer, der overholder den
  - at vi kan bygge et heap i $\Theta(n)$ fra et uorganiseret array med Build-Max-Heap

# Dagens temaer

## Opsummering

- Vi har mødt basale datastrukturer som stacks, queues og linked lists
  - Stacks følger LIFO ('last in, first out')
  - Queues følger FIFI ('first in, first out')
  - Linked lists erstatter array-fundamentet med pointers fra hvert element til det næste (og til det forrige, hvis det er en doubly linked list)
- Vi har mødt heaps og lært
  - at et heap, der kan fortolkes som et binært træ
  - at et max-heap skal overholde heap-egenskaben: 'For alle $i > 1$ gælder $A[\text{Parent}(i)] \geq A[i]$'
  - at vi med Max-Heapify i $\Theta(\log n)$ tid kan re-etablere heap-egenskaben i en knude $A[i]$, så længe Left($i$) og Right($i$) er rødder i træer, der overholder den
  - at vi kan bygge et heap i $\Theta(n)$ fra et uorganiseret array med Build-Max-Heap
- Vi har også set på to applikationer for heaps
  - Vi kan sortere in-place og i $\Theta(n \log n)$ med heapsort, der udnytter heap-strukturen og de logaritmiske operationer
  - Vi kan implementere en priority queue med heaps og understøtte insertion og deletion i $\Theta(\log n)$ tid

# Tak for i dag!

## Flere exercises..

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