

analyse van algoritmen

datastructuren en algoritmen
2011 09 08
college 2

- niet meten na, maar voorspellen voor implementatie
- running time van een algoritme als functie van de grootte van de input
- grote-Oh notatie
- belangrijke klassen:
constant, logaritmisch, lineair, $n \log n$, kwadratisch, polynomiaal, exponentieel

schema

- recap
- lineaire datastructuren
- hierarchische datastructuren
- materiaal

schema

- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
- materiaal

schema

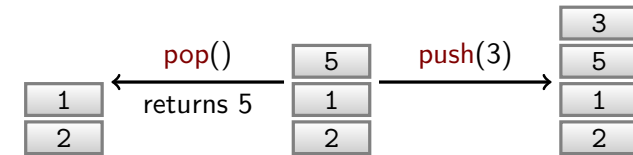
- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
 - bomen
- materiaal

stacks: voorbeelden

- stapel borden
- rijtje stappen waar je 'undo' op kunt toepassen in een editor
- rijtje pagina's waar je 'back' op kunt toepassen in een browser
- method stack in de Java Virtual Machine

stacks: eigenschappen

- **LIFO** = last-in first-out
- welke data doet er niet toe
- operaties: item erbij (**push**), item eraf (**pop**)



voorbeeld: method stack in de Java Virtual Machine

actieve methods, met local variables en een program counter

```
main() {  
    int i = 5;  
    foo(i);  
}  
  
foo(int j) {  
    int k;  
    k = j+1;  
    bar(k);  
}  
  
bar(int m) {  
    ...  
}
```

```
bar  
PC = 1  
m = 6
```

```
foo  
PC = 3  
j = 5  
k = 6
```

```
main  
PC = 2  
i = 5
```

Abstract Data Type (ADT)

we geven datastructuren als ADTs

een ADT specificceert

- welke data wordt opgeslagen?
- wat zijn de operaties op de data?
- wanneer krijg je een error?

ADT voor stacks: errors

- aanroepen van `pop()` op een lege stack
- aanroepen van `top()` op een lege stack

ADT voor stacks

main operations:

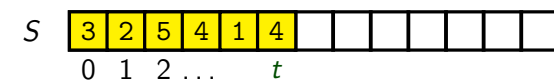
- `push(o)`
voegt o toe aan de top van een stack
- `pop()`
verwijdert laatst toegevoegde element en retourneert dit

auxiliary operations:

- `size()`
geeft het aantal elementen van de stack
- `isEmpty()`
geeft boolean die aangeeft of de stack leeg is
- `top()`
geeft het laatst toegevoegde element (zonder dit te verwijderen)

stacks: implementatie met array

- de grootte N wordt van tevoren bepaald
- elementen worden van links naar rechts toegevoegd
- variabele t wijst naar de top van de stack
- initieel: $t = -1$ oftewel de stack is leeg



stacks: implementatie met array

Algorithm push(o):

```
if size() =  $N$  then
    throw FullStackException
else
     $t := t + 1$ 
     $S[t] := o$ 
```

Algorithm pop():

```
if size() = 0 then
    throw EmptyStackException
else
     $o := S[t]$ 
     $t := t - 1$ 
    return  $o$ 
```

schema

- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
 - bomen
- materiaal

stacks: implementatie met array

- operaties zijn allemaal in $\mathcal{O}(1)$
- maximum grootte is van tevoren bepaald (niet conform ADT)
- element toevoegen aan stack van maximale grootte levert error (niet conform ADT)

queues: eigenschappen

- FIFO = first-in first-out
- welke data doet er niet toe
- operaties: item erbij (**enqueue**) item eraf (**dequeue**)

queues: voorbeelden

- postkantoor
- printer-queue
- multiprogramming

ADT voor queues: errors

- aanroepen van `dequeue()` op een lege queue
- aanroepen van `front()` op een lege queue

ADT voor queues

main operations:

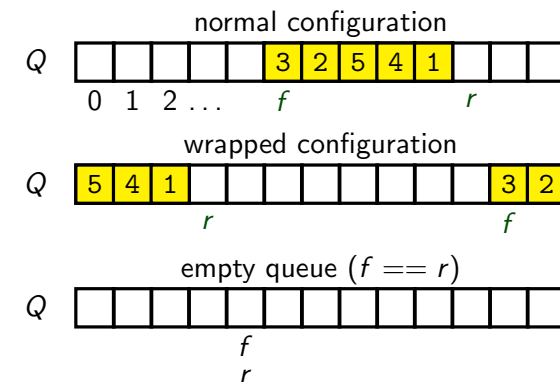
- `enqueue(o)`
voegt element o toe aan het eind van de queue
- `dequeue()`
verwijdert het object vooraan in de queue en retourneert dit

auxiliary operations:

- `size()`
geeft de grootte van de queue
- `isEmpty()`
geeft aan of de queue leeg is
- `front()`
geeft het eerste element van de queue (zonder dit te verwijderen)

queues: implementatie met circulair array

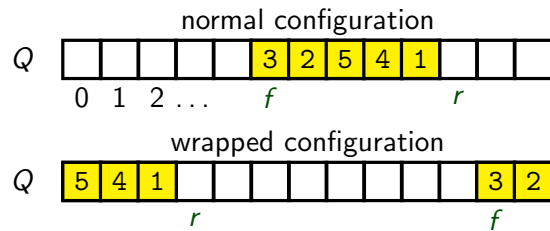
- bepaal van tevoren de grootte $N - 1$
- gebruik array ter grootte N , circulair
- f : index van eerste element
 r : index direct na laatste element (daar staat dus niets)
- initieel: $f = r = 0$



queues: implementatie met circulair array

Algorithm enqueue(o):

```
if size() =  $N - 1$  then
    throw FullQueueException
else
     $Q[r] := o$ 
     $r := (r + 1) \bmod N$ 
```



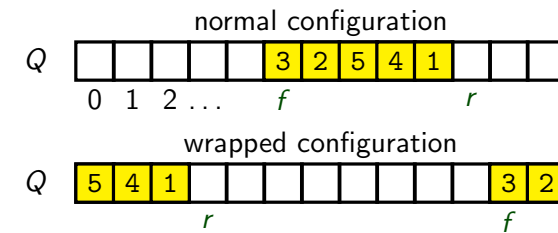
queues: implementatie met circulair array

Algorithm size():

```
return  $(N + r - f) \bmod N$ 
```

Algorithm isEmpty():

```
return size() = 0
```



queues: implementatie met circulair array

- operaties zijn allemaal in $\mathcal{O}(1)$
- maximum grootte is van tevoren bepaald (niet conform ADT)
- element toevoegen aan queue van maximale grootte levert error (niet conform ADT)

schema

- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
 - bomen
- materiaal

vectoren: eigenschappen

- lineaire datastructuur
net als bij stacks en queues
- welke data doet er niet toe
net als bij stacks en queues
- elementen opvragen op alle posities via rank/index
anders dan stacks en queues

vectoren: implementatie met array

- array A van grootte N
- $A[i]$ bevat het element met rank i
variabele n geeft aantal elementen aan
- makkelijk te implementeren:
 $size()$, $isEmpty()$, $elemAtRank(r)$, $replaceAtRank(r, o)$
- moeilijker te implementeren (opschuiven !):
 $insertAtRank(r, o)$, $removeAtRank(r)$

vectoren: ADT

n elementen; rank loopt van 0 tot en met $n - 1$
errors zijn weggelaten !

accessor operations:

- $elemAtRank(r)$

update operations:

- $replaceAtRank(r, o)$
- $insertAtRank(r, o)$
- $removeAtRank(r)$

generic operations:

- $size()$
- $isEmpty()$

insertAtRank: implementatie met array in $\mathcal{O}(n)$

Algorithm $insertAtRank(r, o)$:

```
for  $i = n - 1$  downto  $r$  do
     $A[i + 1] := A[i]$ 
 $A[r] := o$ 
 $n := n + 1$ 
```

beste als $r = n - 1$

slechtste als $r = 0$

iha $n - r + 1$ opschuiven

schema

- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
 - bomen
- materiaal

lijsten: ADT (1)

accessor operations:

- `first()`
- `last()`
- `before(n)`
- `after(n)`

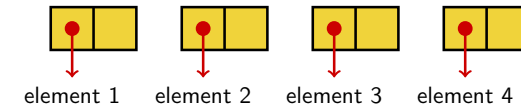
query operations:

- `isFirst(n)`
- `isLast(n)`

(n is een knoop)

lijsten: eigenschappen

- lineaire datastructuur
- welke data doet er niet toe
- knopen met toegang tot vorige en volgende



lijsten: ADT (2)

generic operaties:

- `size()`
- `isEmpty()`

update operaties:

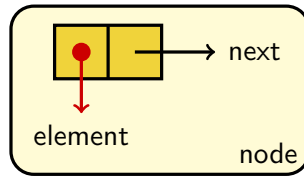
- `replaceElement(n, o)`
- `swapElements(n, m)`
- `insertBefore(n, o)`
- `insertAfter(n, o)`
- `insertFirst(o)`
- `insertLast(o)`
- `remove(n)`

(n en m zijn knopen; o is data)

knopen in een singly linked list

een knoop bevat:

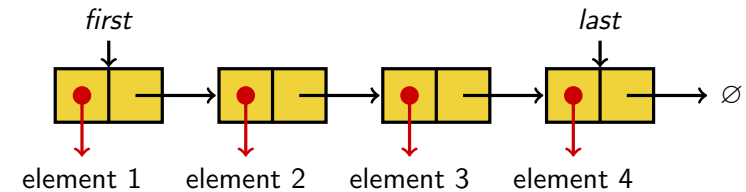
- een element (of: data)
- een link (geen methode) *next* naar de volgende node



lijsten: implementatie met singly linked lists

voor de hele lijst:

- variabele *first* wijst naar de eerste knoop
- eventueel:
variabele *last* wijst naar de laatste knoop
- eventueel:
variabele *size* geeft het aantal knopen aan



size: vergelijk twee mogelijkheden

we hebben een variabele *size*: in $\mathcal{O}(1)$

```
Algorithm size():  
    return size
```

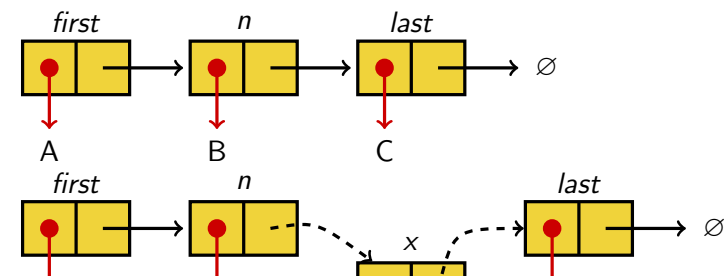
we hebben die niet: in $\mathcal{O}(n)$

```
Algorithm size():  
    s := 0  
    m := first  
    while m ≠ ∅ do  
        s := s + 1  
        m := m.next  
    return s
```

insertAfter in $\mathcal{O}(1)$

Algorithm insertAfter(*n*, *o*):

```
new node x  
x.element := o  
x.next := n.next  
n.next := x  
if n = last then  
    last := x  
size := size + 1
```



lijsten met singly linked lists: overzicht

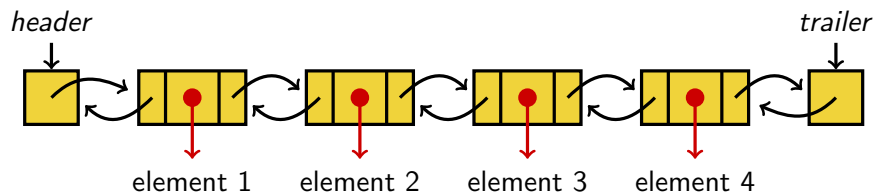
Operation	Worst case Complexity
size, isEmpty	$\mathcal{O}(1)$
first, last, after	$\mathcal{O}(1)$
before	$\mathcal{O}(n)$
replaceElement, swapElements	$\mathcal{O}(1)$
insertFirst, insertLast	$\mathcal{O}(1)$
insertAfter	$\mathcal{O}(1)$
insertBefore	$\mathcal{O}(n)$
remove	$\mathcal{O}(n)$
atRank, rankOf, elemAtRank	$\mathcal{O}(n)$
replaceAtRank	$\mathcal{O}(n)$
insertAtRank, removeAtRank	$\mathcal{O}(n)$

NB: size, last, insertLast, remove

lijsten: implementatie met doubly linked list

voor de hele lijst:

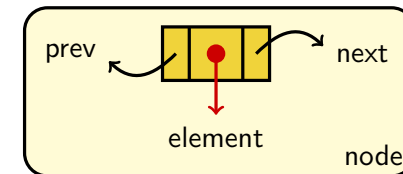
- variabele *header*
- variabele *trailer*
- eventueel: variabele *size* geeft het aantal knopen aan



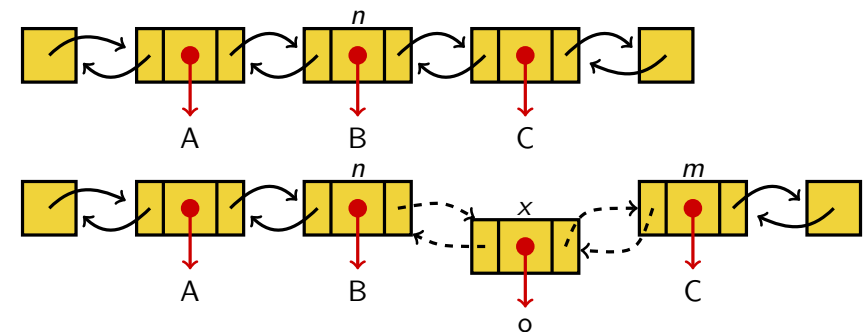
knopen in een doubly linked list

een knoop bevat:

- een element (of: data)
- een link naar de volgende knoop
- een link naar de vorige knoop



insertAfter met doubly linked list in $\mathcal{O}(1)$



lijsten met doubly linked list: overzicht

Operation	Worst case Complexity
size(), isEmpty()	$\mathcal{O}(1)$
first, last, after	$\mathcal{O}(1)$
before	$\mathcal{O}(1)$
replaceElement(,,) swapElements	$\mathcal{O}(1)$
insertFirst(), insertLast()	$\mathcal{O}(1)$
insertAfter(,)	$\mathcal{O}(1)$
insertBefore(,)	$\mathcal{O}(1)$
remove()	$\mathcal{O}(1)$
(nog een) rankOf(), elemAtRank()	$\mathcal{O}(n)$
replaceAtRank(,)	$\mathcal{O}(n)$
insertAtRank(,,) removeAtRank()	$\mathcal{O}(n)$

rijtje of sequence

combinatie van vector en lijst

toegang tot een element via rank/index
en ook via volgende/vorige

schema

- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
 - bomen
- materiaal

vectoren, lijsten, rijtjes

het lijkt allemaal erg op elkaar;
ook in een lijst zou je van rank kunnen spreken

belangrijk: welke access operaties heb je

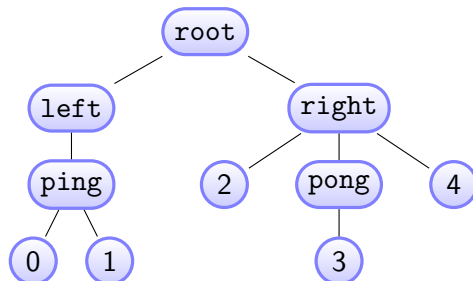
iterator

loopt door een lineaire datastructuur heen

- `object()`
geeft het current object terug
- `hasNext()`
geeft aan of er nog meer objecten zijn
- `nextObject()`
gaat naar het volgende object en geeft dit terug

bomen: eigenschappen

- verzameling knopen met ouder-kinder relatie
- unieke wortel
- elke niet-wortel knoop heeft unieke ouder



schema

- recap
- lineaire datastructuren
- hierarchische datastructuren
 - bomen
- materiaal

bomen: voorbeelden

- directory
- boek
- organisatiestructuur

bomen: terminologie

- wortel
- ouder, kind
- voorganger, opvolger
- interne knoop, externe knoop (blad)
- diepte van een knoop, hoogte van een boom
- sub-boom

ADT voor bomen: generic operations

- `size()`
geeft het aantal knopen
- `elements()`
geeft de verzameling van alle elementen die op knopen staan
- `positions`
geeft de verzameling van alle knopen
- `swapElements(v , w)`
verwisselt elementen op knopen v en w
- `replaceElement(v , e)`
vervangt element op knoop v door e

ADT voor bomen: accessor operations

posities = knopen

- `root()`
geeft de (unieke) wortel van de boom
- `children(v)`
geeft de lijst van de (geordende) kinderen van knoop v
- `parent(v)`
geeft de (unieke) ouder van knoop v
(`error` als v de wortel is)

ADT voor bomen: generic operations

- `isInternal(v)`
- `isExternal(v)`
- `isRoot(v)`

diepte en hoogte

diepte van een knoop:

aantal voorgangers oftewel lengte van pad naar de wortel

hoogte van een boom:

grootste diepte

preorder traversal

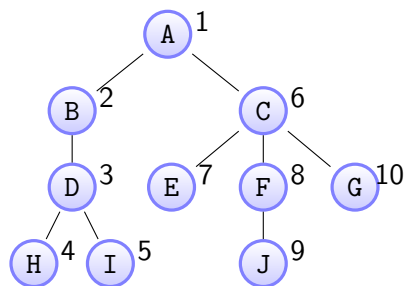
bezoek eerst de knoop en dan zijn opvolgers

Algorithm preOrder(v):

visit(v)

for each w in children(v) **do**

preOrder(w)



traversal

hoe bezoek je alle knopen precies één keer?

hoe efficiënt is preorder?

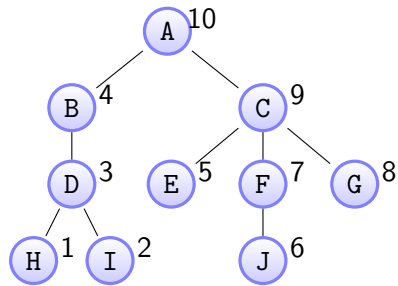
op elke knoop: 1 plus het aantal kinderen

dus totaal in $\mathcal{O}(n)$ met n het aantal knopen

postorder

bezoek eerst alle opvolgers en dan de knoop zelf

Algorithm postOrder(v):
 for each w in children(v) **do**
 postOrder(w)
 visit(v)



hoe efficient is postorder?

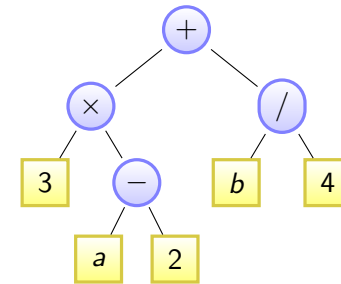
in $\mathcal{O}(n)$

binaire bomen

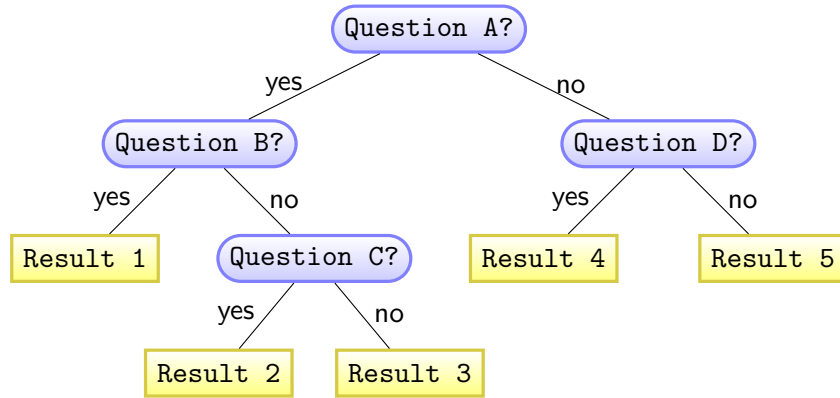
elke knoop heeft 0 of 2 (links en rechts) opvolgers

binaire bomen voor arithmetische expressie

$$(3 \times (a - 2)) + (b/4)$$



binaire beslissingsboom



binaire bomen: eigenschappen

h : hoogte
 e : aantal externe knopen
 i : aantal interne knopen

- $e \geq i + 1$
- $e \leq 2^h$
- $e = i + 1$

(Zie Theorem 2.8 van het boek.)

ADT voor binaire bomen

specialisatie van ADT voor bomen met extra:

- $\text{leftChild}(v)$
- $\text{rightChild}(v)$
- $\text{sibling}(v)$

pre- en postorder voor binaire bomen

we kunnen de algemene methoden specifieker opschrijven

Algorithm $\text{binaryPreOrder}(v)$:

```
visit(v)
if isInternal(v) then
    binaryPreOrder(leftChild(v))
    binaryPreOrder(rightChild(v))
```

Algorithm $\text{binaryPostOrder}(v)$:

```
if isInternal(v) then
    binaryPostOrder(leftChild(v))
    binaryPostOrder(rightChild(v))
visit(v)
```

inorder traversal

eerst linker sub-boom, dan de knoop zelf, dan zijn rechter sub-boom

```
inOrder(v):  
  if isInternal(v) then  
    inOrder(leftChild(v))  
  visit(v)  
  if isInternal(v) then  
    inOrder(rightChild(v))
```

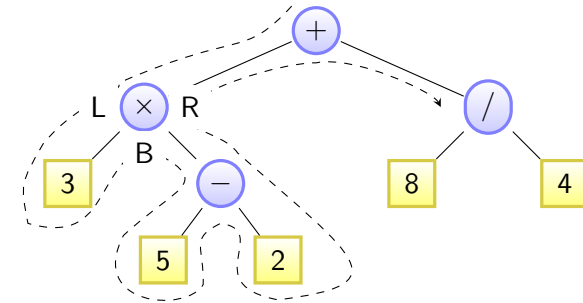
Euler traversal

instantieer visitLeft, visitBelow, visitRight zoals gewenst

```
eulerTour(v):  
  visitLeft(v)  
  if isInternal(v) then  
    eulerTour(leftChild(v))  
  visitBelow(v)  
  if isInternal(v) then  
    eulerTour(rightChild(v))  
  visitRight(v)
```

Euler traversal

generieke beschrijving van traversals



implementaties van bomen

- binaire bomen met vectoren
gebruik de level-numbering
positions en elements zijn in $\mathcal{O}(n)$
- binaire bomen met linked lists
algemene bomen met linked lists
positions en elements zijn in $\mathcal{O}(n)$

schema

- recap
- lineaire datastructuren
 - stacks
 - queues
 - vectoren
 - lijsten
 - singly linked lists
 - doubly linked lists
 - rijtjes
- hierarchische datastructuren
 - bomen
- materiaal

materiaal

- boek 2.1, 2.2, 2.3

extra materiaal

- wiki over dequeues
- wiki over Leonhard Euler
- wiki over binaire bomen