Stacks and queues

Hoofdstuk 6

Stacks and queues

INTRODUCTIE

In dit hoofdstuk worden drie datastructuren *stack*, *queue* en *deque* behandeld. Om deze datastructuren te implementeren, worden onder andere arrays en linked lists gebruikt. Het is van belang dat u tijdens het bestuderen van het hoofdstuk datastructuren en implementaties daarvan goed uit elkaar houdt.
Het volledige hoofdstuk 6 van het tekstboek behoort tot de leerstof.

LEERDOELEN
Na het bestuderen van dit hoofdstuk wordt verwacht dat u
- de volgende datastructuren in eigen woorden kunt beschrijven: stapel, wachtrij en deque
- de verschillende implementatievormen van genoemde datastructuren uiteen kunt zetten
- ADT's voor stapels, wachtrijen en deque's kunt specificeren
- de verschillen in complexiteit kunt verklaren tussen operaties op stapels, wachtrijen en deques die geïmplementeerd zijn met arrays, enkel- en dubbelgeschakelde lijsten alsmede de verschillen in de ruimtecomplexiteit van aldus geïmplementeerde datastructuren
- het ontwerppatroon adapter kunt omschrijven.

Studeeraanwijzingen
In de bijlage vindt u diagrammen van de belangrijkste interfaces en klassen die worden besproken in dit hoofdstuk. Een klasse waarvan de naam niet expliciet in het tekstboek wordt genoemd, maar waarvan wel pseudocode is gegeven of waarvan de implementatie van enkele methoden is gegeven, herkent u aan het afwijkende lettertype in een diagram.

LEERKERN

6.1 Stacks

6.1.1 THE STACK ABSTRACT DATA TYPE

In deze paragraaf wordt de stapel als datastructuur besproken. De ADT Stack wordt in het tekstboek gegeven in paragraaf 6.1.1 en in codefragment 6.1 staat de interface Stack die deze ADT in Java specificeert. Merk op dat de interface generiek is gespecificeerd. In de codefragmenten 6.2 en 6.4 uit het tekstboek staan twee verschillende klassen die de interface Stack implementeren. U vindt in deze klassen de methoden terug die in de ADT beschreven zijn en waarvan de signaturen in de interface Stack gespecificeerd zijn.

Deze werkwijze zal in de hele cursus gevolgd worden: een datastructuur wordt door middel van een ADT beschreven, vervolgens met een Java-interface-taalconstructie gespecificeerd en door middel van klassen op verschillende manieren geïmplementeerd.

In deze paragraaf wordt ook de 'built-in-'klasse java.util.Stack vermeld. Omdat een stapel een veelgebruikte datastructuur is, is deze klasse standaard als klasse in Java aanwezig Deze klasse is een subklasse van de standaardklasse Vector. Omdat alle methoden van Vector worden geërfd, is het mogelijk om ook ergens midden in het Stack-object een nieuw element toe te voegen, te verwijderen of te inspecteren, wat natuurlijk niet de bedoeling is bij een stack.

6.1.2 A SIMPLE ARRAY-BASED STACK IMPLEMENTATION

De volgende notatie wordt in het tekstboek gebruikt: *N* is de maximale ruimte gereserveerd voor een array en *n* is het aantal elementen bewaard in de datastructuur.
Let op: in het tekstboek wijkt de notatie soms af van wat u gewend bent in de voorgaande Java-cursussen. Namen van attributen en variabelen beginnen niet altijd met een kleine letter en een array zoals Object[] S wordt soms genoteerd als Object S[].

Codefragment 6.2 geeft een stack-implementatie die op een array is gebaseerd. We lichten een deel van deze implementatie toe.
Het is in Java toegestaan om een array van een geparametriseerd type te *declareren*, maar het is niet toegestaan om een dergelijke array ook te *creëren*. Daarom wordt een array van type Object gecreëerd en wordt er een cast aan toegevoegd:

```
data = (E[]) new Object[capacity];
               //safe cast;compiler may give warning
```

De compiler zal hier een waarschuwing geven. Deze luidt ongeveer als volgt: "Type safety: Unchecked cast from Object[] to E[]". Bij creatie van een stack wordt een actuele type parameter opgegeven en alleen elementen van dit type kunnen met methode push op de stack geplaatst worden. Er worden dus altijd elementen van het juiste type toegevoegd en de cast zal dan ook geen problemen opleveren.

Voorbeeld

We creëren twee stacks, één die alleen gehele getallen kan bevatten, en één die alleen Strings kan bevatten, en voegen aan beide stacks enkele elementen als volgt toe:

```
ArrayStack<Integer> intstack = new ArrayStack<Integer>();
ArrayStack<String> stringstack = new ArrayStack<String>();
intstack.push(new Integer(3));
intstack.push(new Integer(-1));
intstack.push("aap"); // dit geeft een compiler fout!!
stringstack.push("aap");
stringstack.push("noot");
stringstack.push(new Integer(3)); // compiler fout!!
```

OPGAVE 6.1
Maak opgave R-6.3 uit het tekstboek.

OPGAVE 6.2
Hoe zou u een eigen klasse MijnStack kunnen ontwerpen die ook gebruikmaakt van een Vector maar die niet het nadeel heeft van de klasse Stack uit het package java.util? Geef een implementatie van deze klasse.

6.1.4 MATCHING PARENTHESES AND HTML TAGS

In deze paragraaf worden twee voorbeelden van het gebruik van een stack gegeven. Het tweede voorbeeld behoort niet tot de leerstof.

6.2 Queues

6.2.2 A SIMPLE ARRAY-BASED IMPLEMENTATION

De klasse ArrayQueue van codefragment 6.8 is een implementatie van een wachtrij. Deze implementatie van de wachtrij maakt gebruik van een circulaire array van grootte *capacity*.
In tabel 6.1 ziet u diverse operaties op een circulaire array van Integers van grootte 4.

TABEL 6.1 Operaties op een circulaire array van grootte 4

operatie	*waarde van f ,(avail) en sz*	*array*
	f=0, sz=0	| | | | |
enqueue(2)	f=0, avail=0, sz=1	| 2 | | | |
enqueue(4)	f=0, avail=1, sz=2	| 2 | 4 | | |
dequeue()	f=1, sz=1	| | 4 | | |
enqueue(1)	f=1, avail=2, sz=2,	| | 4 | 1 | |
enqueue(5)	f=1, avail=3, sz=3	| | 4 | 1 | 5 |
dequeue()	f=2, sz=2	| | | 1 | 5 |
enqueue(3)	f=2, avail=0, sz=3	| 3 | | 1 | 5 |
dequeue()	f=3, sz=2	| 3 | | | 5 |
dequeue()	f=0, sz=1	| 3 | | | |
dequeue()	f=1, sz=0	| | | | |

OPGAVE 6.3
Beschrijf hoe de queue ADT geïmplementeerd kan worden met behulp van 2 stapels. Wat is in dit geval de verwerkingstijd van de operaties enqueue en dequeue?

6.2.3 IMPLEMENTING A QUEUE WITH A SINGLY LINKED LIST

OPGAVE 6.4
Maak opgave C-6.19 uit het tekstboek. Wat is de verwerkingstijd van uw algoritme?

6.3 **Double-ended queues**

OPGAVE 6.5
De klasse ArrayDeque implementeert de ADT Deque met behulp van een niet-circulaire array van grootte *N*. Geef voor elke methode van ArrayDeque een algoritme in pseudo-code.

TERUGKOPPELING

Uitwerking van de opgaven

6.1 Het bijzondere probleem is een lege stack, het algemene probleem is op te lossen door het bovenste element van de stack te halen en vervolgens methode clear aan te roepen op de kleinere stack:

```
public void clear(Stack s) {
  if (s.isEmpty() {
    // doe niets
  }
  else {
    s.pop();
    clear(s);
  }
}
```

6.2 We moeten hier geen overerving toepassen maar compositie. Klasse MijnStack is geen Vector, maar heeft een Vector (als attribuut), en kent een beperkt aantal methodes die geïmplementeerd worden door de overeenkomstige methode op het attribuut van type Vector aan te roepen.

```
import java.util.Vector;

public class MijnStack<E> {
  private Vector<E> vector = null;

  public MijnStack() {
    vector = new Vector<E>();
  }

  public void push(E e) {
    vector.addElement(e);
  }

  public E pop(){
    if (isEmpty()) {
      return null;
    }
    E e = vector.elementAt(size() - 1);
    vector.removeElementAt(size() - 1);
    return e;
  }

  public int size() {
    return vector.size();
  }

  public boolean isEmpty() {
    return vector.isEmpty();
  }

  public E top(){
    if (isEmpty()) {
      return null;
    }
    return vector.elementAt(size() - 1);
  }
}
```

6.3 De ADT Queue wordt geïmplementeerd met behulp van de twee stapels S en S'. Voor de operatie enqueue wordt het element op de stapel S geplaatst. De complexiteit van deze operatie komt overeen met de complexiteit van de operatie push. De complexiteit van methode push is voor beide stackimplementaties (met een array en met een geschakelde lijst) $O(1)$.
Voor de operatie dequeue worden eerst alle elementen van de stapel S gehaald met de operatie pop en op de stapel S' geplaatst met de operatie push. Dan wordt de top van S' – en dat is het laatste element van S – van die stapel gehaald met een operatie pop. Daarna worden alle resterende elementen van S' gehaald met de operatie pop en op S teruggeplaatst met de operatie push.
Per element worden dus de volgende operaties uitgevoerd: S.pop, S'.push, S'.pop en S.push. Deze vier operaties zijn elk $O(1)$, onafhankelijk van de gekozen stackimplementatie.
Daardoor is de complexiteit van de operatie dequeue $O(n)$, zelfs $\Theta(n)$, waarbij n het aantal elementen van de wachtrij is.

6.4 Haal een element van de stapel S, vergelijk het element met de gezochte x
Als het element niet het gezochte is, plaats dan het element in de wachtrij en herhaal dit totdat x is gevonden of totdat S leeg is.
Nu is bekend of x voorkwam in S, maar S zelf is ook gewijzigd en moet nog in de oorspronkelijke staat gebracht worden.
Vraag daarom de grootte van de wachtrij Q op en bewaar deze in een variabele m. Zet alle elementen van Q terug op S. De elementen die terug op de stapel zijn geplaatst, staan nu in de verkeerde volgorde. Haal daarom de eerste m elementen nogmaals van de stapel S, zet ze in de wachtrij Q en zet ze daarna weer terug op S. Ze staan dan in de juiste volgorde.

In het slechtste geval wordt het element x niet gevonden en worden alle n elementen één voor één uit S gehaald en in Q geplaatst, daarna worden ze één voor één uit Q gehaald en op S geplaatst, waarna de hele procedure zich nog één keer herhaalt. Elk element wordt in totaal dus twee keer van de stapel gehaald, twee keer op de stapel gezet, twee keer in de wachtrij gezet en twee keer uit de wachtrij gehaald. Methoden push, pop, enqueue en dequeue zijn allemaal $O(1)$ (dit is onafhankelijk van de implementatie van de stapel en de wachtrij). De complexiteit is dan ook $O(n)$.
We kunnen hier niet spreken van $\Theta(n)$, omdat het algoritme wel sneller uitgevoerd kan worden; dit is het geval als het element snel wordt gevonden.

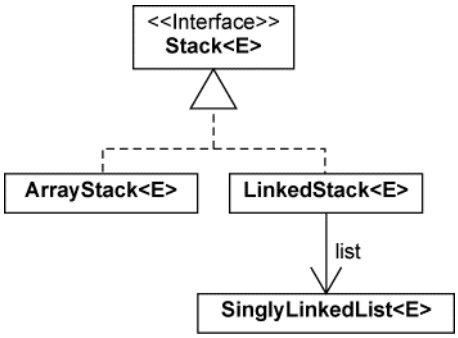
6.5 De klasse ArrayDeque heeft twee attributen: een array *D* van grootte *N* die de elementen van de deque bewaart, en de lengte van de deque size.

```
Algorithm addFirst(e)
    for i ← size – 1 to 0 do
        D[i + 1] ← D[i]
    { end for }
    D[0] ← e
    size++

Algorithm addLast(e)
    D[size] ← e
    size++

Algorithm removeFirst()
    hulp ← D[0]
    for i ← 1 to size – 1 do
        D[i – 1] ← D[i]
    { end for }
    size – –
    return hulp

Algorithm removeLast()
    hulp ← D[size – 1]
    size – –
    return hulp

Algorithm first()
    return D[0]

Algorithm last()
    return D[size – 1]

Algorithm size()
    return size

Algorithm isEmpty()
    return (size == 0)
```

Bijlage

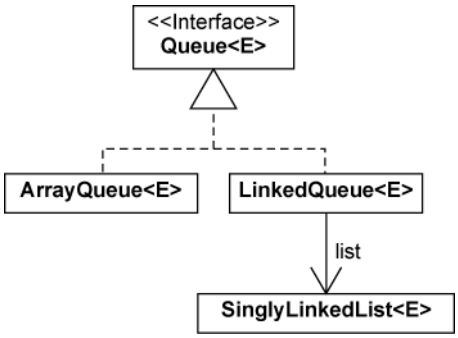
Diagrammen belangrijkste interfaces en klassen

1 Stacks



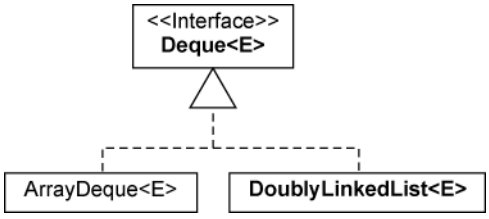
FIGUUR 6.1 Diagram bij ADT Stack

2 Queues



FIGUUR 6.2 Diagram bij ADT Queue

3 Deques



FIGUUR 6.3 Diagram bij ADT Deque