Stack

Hva er en stack?

- En lineær datastruktur der vi til enhver tid kun har tilgang til elementet som ble lagt inn sist
- Et nytt element legges alltid på *toppen* av stakken
- Skal vi ta ut et element, tar vi *alltid* det øverste
- Sammenlikning: Stabel med bøker/tallerkener/mynter
- Kalles også en LIFO-kø (Last In, First Out)

Typiske anvendelser av stack

- Snu rekkefølger (f.eks. "undo" i tekstbehandlere)
- Maskinell lesing og beregning av regneuttrykk
- I kompilatorer, f.eks. til syntakssjekking og *parsing* av programkode
- I operativ- og runtimesystemer, bl.a. for å holde rede på funksjonskall* og bruk av RAM
- Backtracking og simulering av rekursjon

*: "The Automatic Computing Engine", Alan Turing, 1946

Operasjonene på en stack

push: legge til et element øverst på stacken

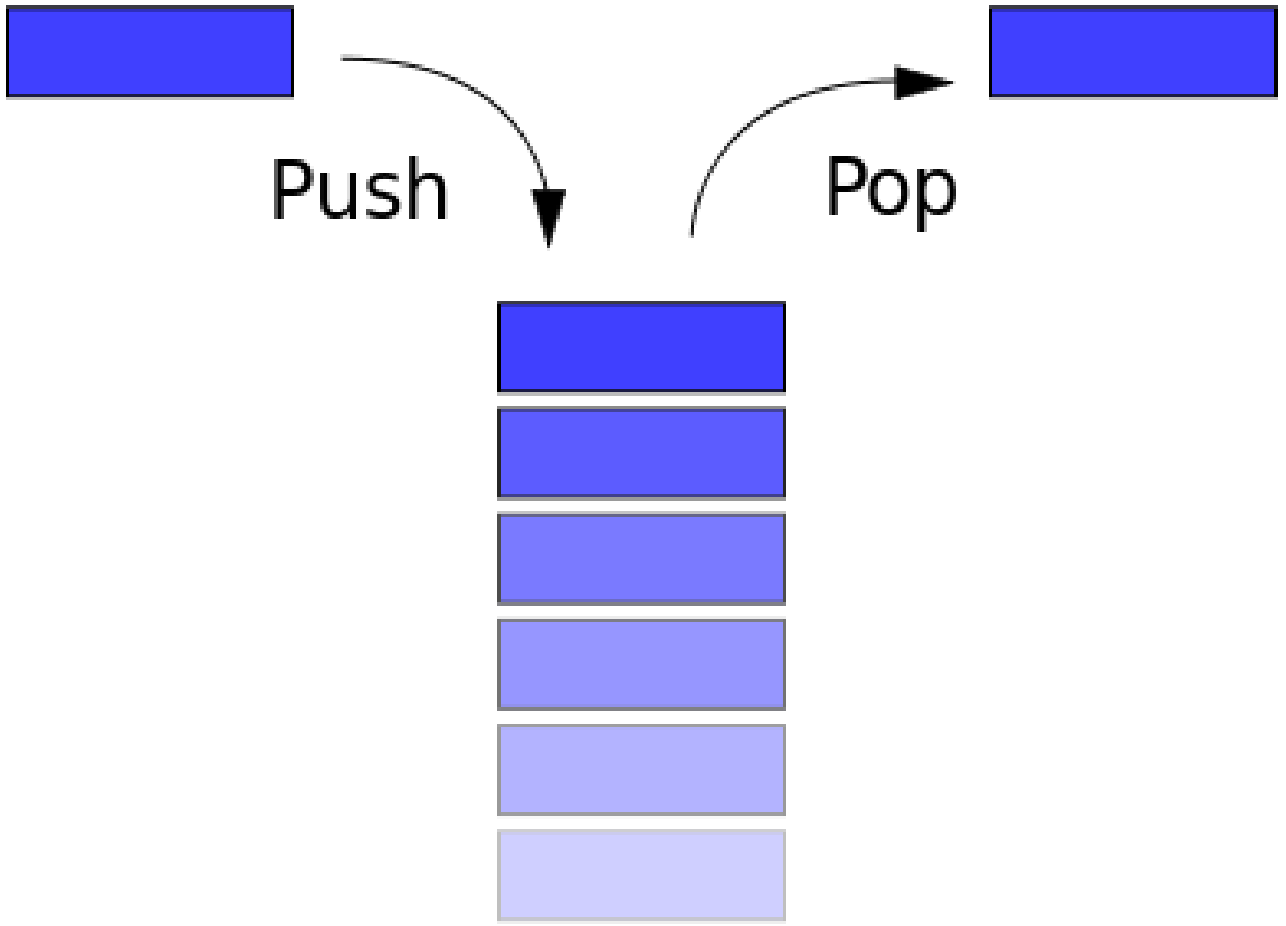
pop: fjerne øverste element

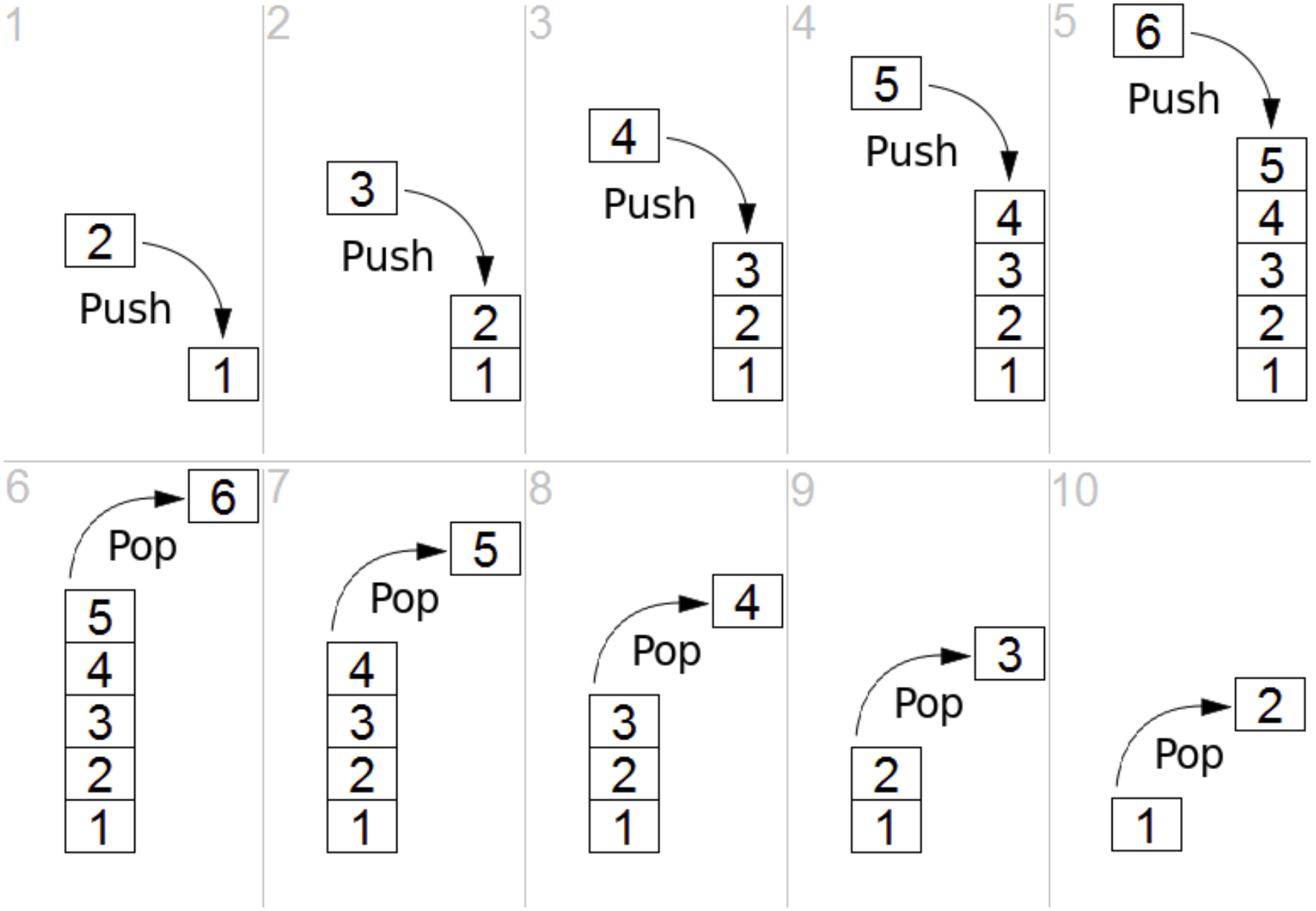
peek: se på/finne øverste element (aka **top**)

isEmpty: sjekk om stacken har null elementer

size: antall elementer som er lagret på stacken

Innsetting og fjerning av data: “push” og “pop”




1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
Pop
6
5
4
3
2
1
7
Pop
5
4
3
2
1
8
Pop
4
3
2
1
9
Pop
3
2
1
10
Pop
2
1

Kallstacken i Java

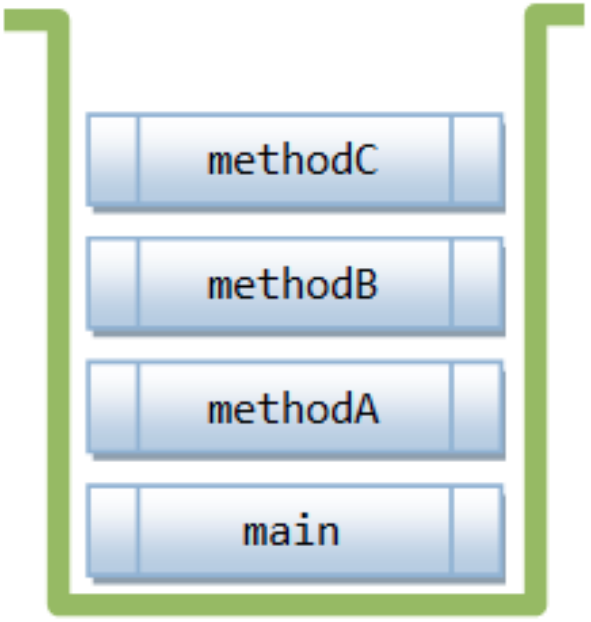
```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit  main()");
   }

   public static void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit  methodA()");
   }

   public static void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit  methodB()");
   }

   public static void methodC() {
      System.out.println("Enter methodC()");
      System.out.println("Exit  methodC()");
   }
}
```

Output:

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit  methodC()
Exit  methodB()
Exit  methodA()
Exit  main()
```

Kallstack

1. JVM invoke the main().
2. main() pushed onto call stack, before invoking methodA().
3. methodA() pushed onto call stack, before invoking methodB().
4. methodB() pushed onto call stack, before invoking methodC().
5. methodC() completes.
6. methodB() popped out from call stack and completes.
7. methodA() popped out from the call stack and completes.
8. main() popped out from the call stack and completes. Program exits.



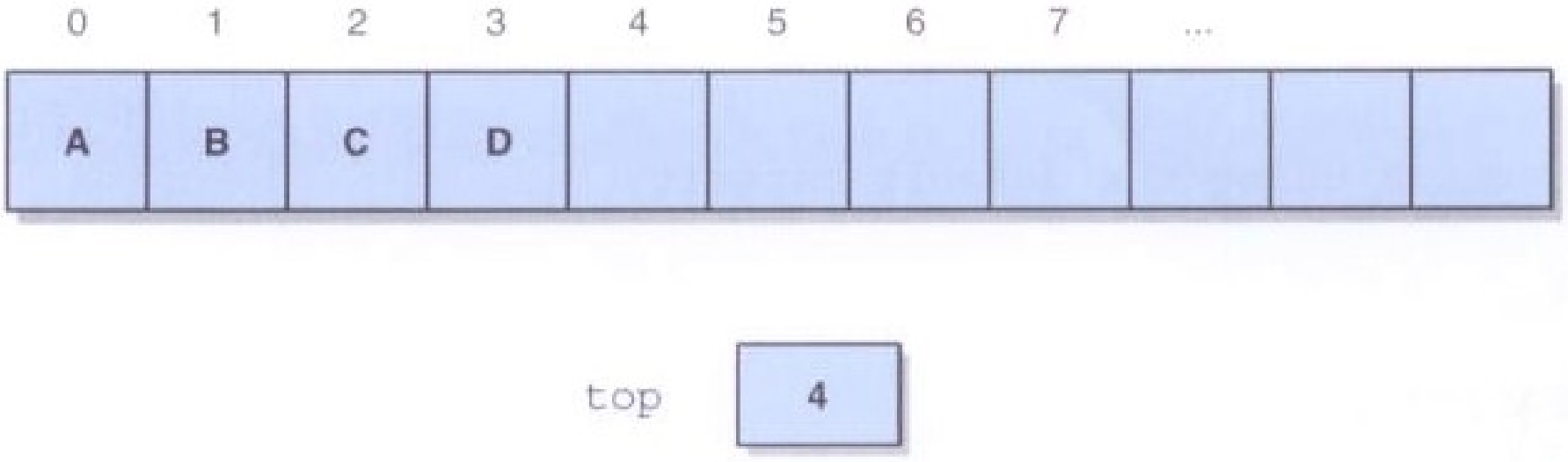
Method Call Stack
(Last-in-First-out Queue)

Stack implementert med array

- Lagrer elementene på stacken i en array med objekter, som initielt er tom
- Holder rede på toppen av stacken med en enkel *teller* som er indeks til neste ledige plass i arrayen
- Teller oppdateres for hver push/pop
- Teller angir antall elementer, stack tom når teller lik 0

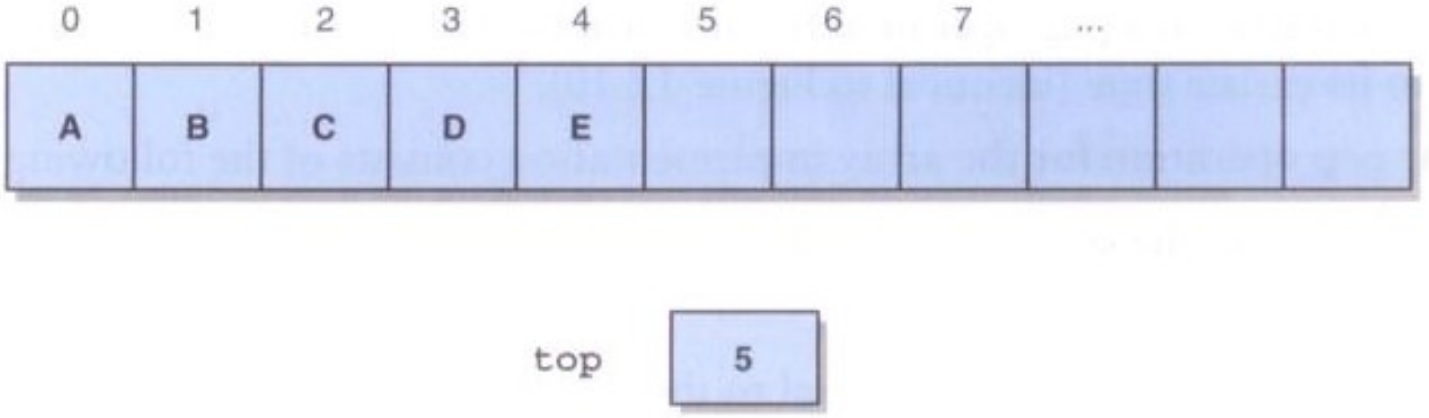
Stack med array

Etter push av verdiene A, B, C og D i rekkefølge på en stack som initielt er tom:

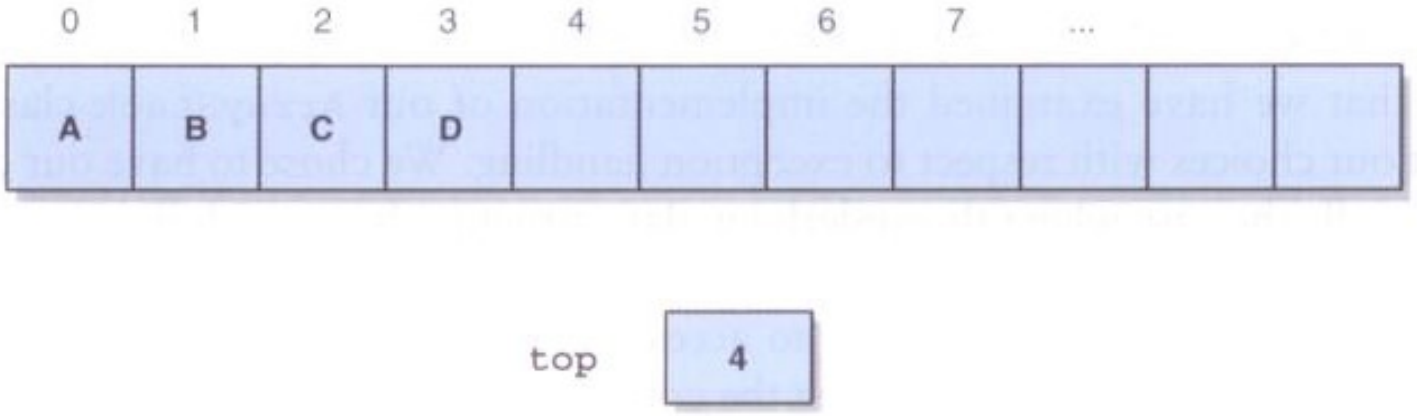


Stack med array: push og pop

Etter push av E:



Etter pop:



Enkel implementasjon av stack der dataene er heltall, uten feilsjekking

- En klasse `IntStack` som inneholder:
  - En array med heltall som lagrer dataene på stacken
  - En teller med antall elementer/første ledige indeks
- Gjør ingen feilsjekking:
  - Crasher hvis arrayen blir full
  - Tåler ikke fjerning av data når stacken er tom
- Se Javakoden

Stack som abstrakt datatype i Java *

```
public interface StackADT<T>
{
    // Legge et nytt element på stacken
    public void push(T element);

    //Fjerne og returnere øverste element på stakken
    public T pop();

    //Returnerer øverste element uten å fjerne det
    public T peek();

    // Antall elementer på stacken
    public int size();

    // Sjekker om stacken er tom
    public boolean isEmpty();
}
```

*: Fra lærebokens stack-modul

Lærebokens implementasjon av stack med array og generisk datatype

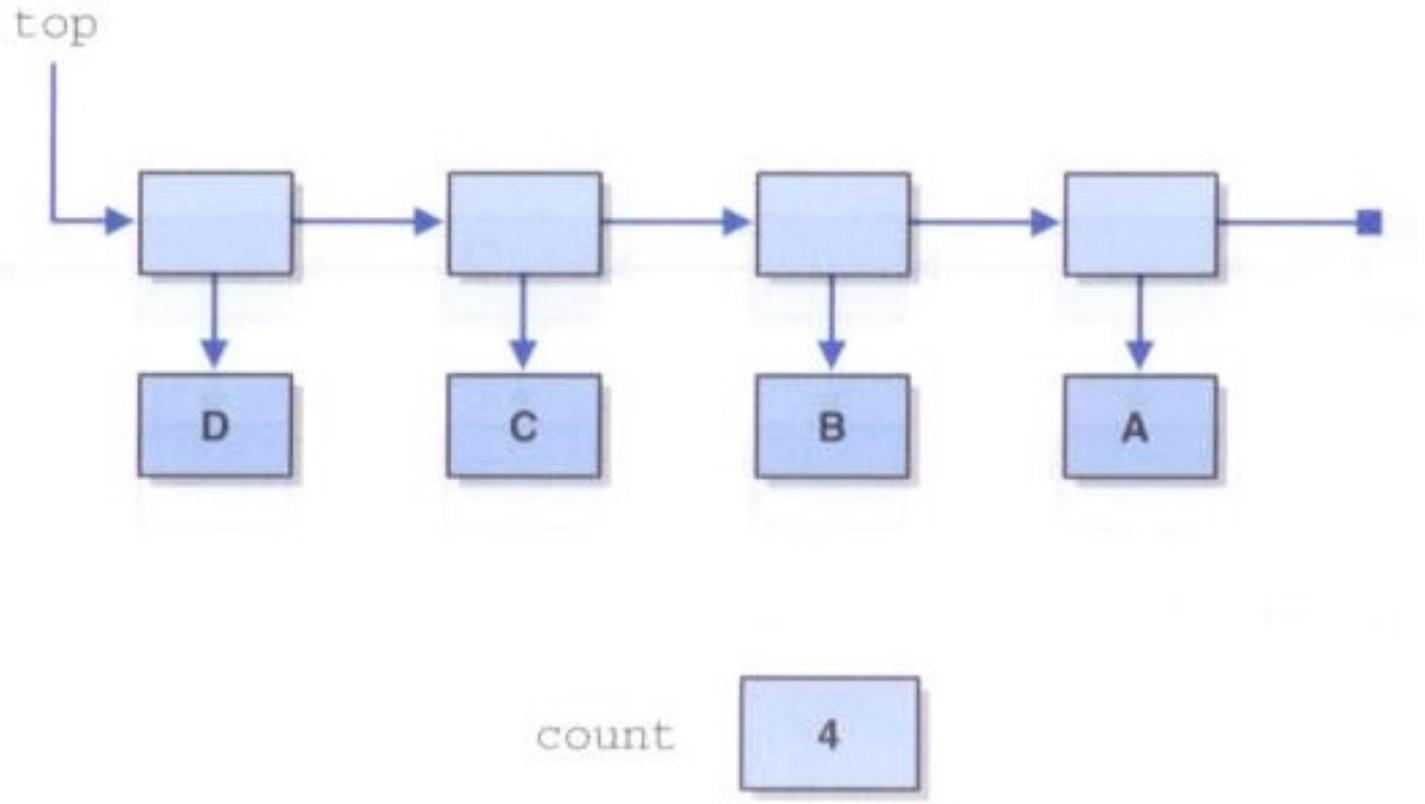
- En klasse `ArrayStack` som inneholder:
  - En array med dataene som er lagret på stack
  - En teller med antall elementer/første ledige indeks
- Arrayen er av en *generisk* datatype: Hva som skal lagres på stacken angis når et `ArrayStack`-objekt opprettes
- Dynamisk stack, øker lengden på array når den er full
- Håndterer feil på en robust måte
- Se implementasjonen fra læreboka

Stack implementert som lenket liste

- Stacken inneholder:
  - Referanse/peker til noden på toppen av stacken
  - En teller med antall elementer
- Nodene inneholder:
  - Referanse/peker til neste element på stacken
  - Referanse/peker til objektet som inneholder dataene
- Se avsnitt 4.6 og implementasjonen fra læreboka

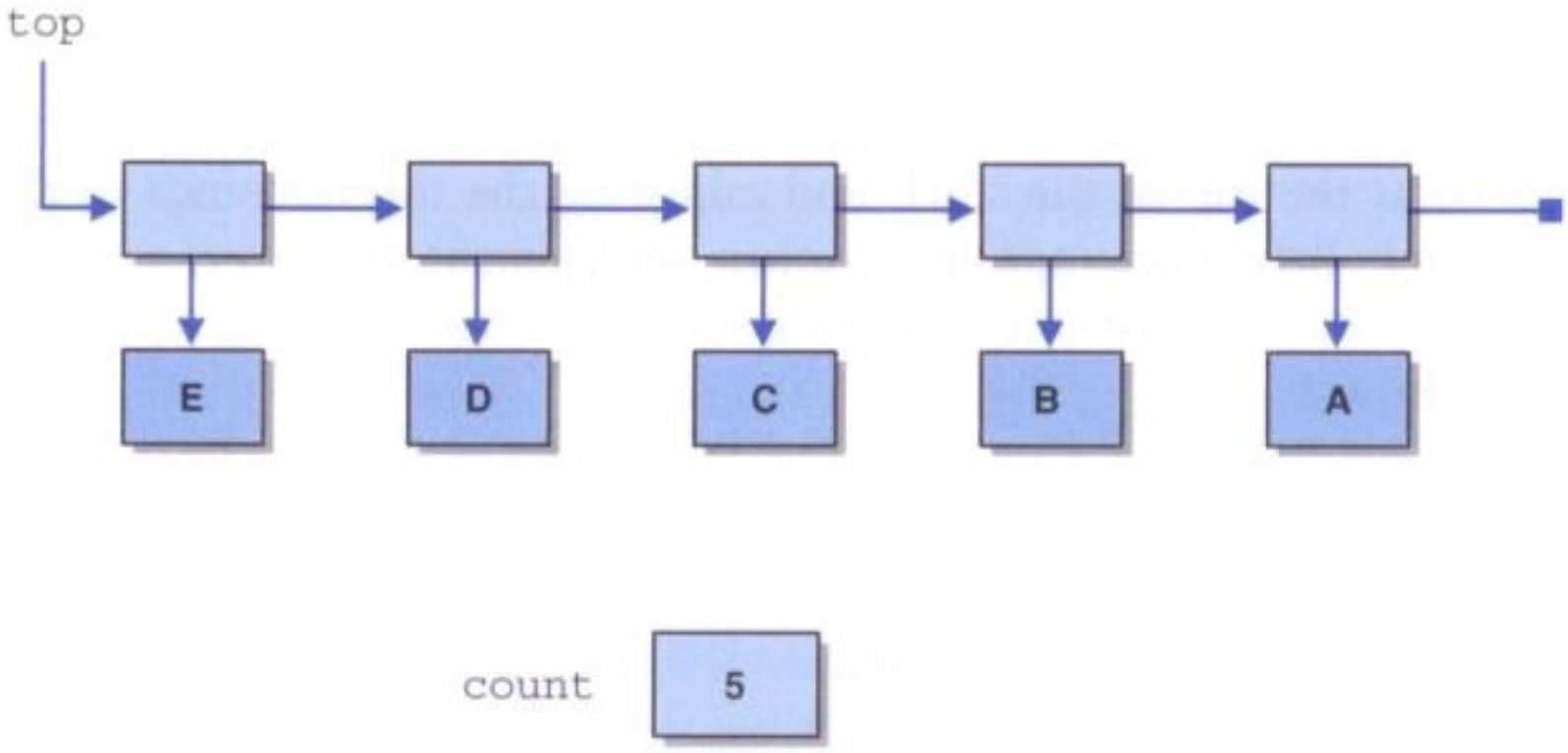
Stack med lenket liste

Etter push av verdiene A, B, C og D i rekkefølge på en stack som initielt er tom:



Stack med lenket liste: push

Etter push av verdien E:



En anvendelse av stack: Skriv ut input i omvendt rekkefølge

- Algoritme:
    1. For hvert tegn i input:
        1.1 Push tegnet på en stack
    2. Inntil stacken er tom:
        2.1 Pop stack og skriv ut tegnet

Eksempel: Omvendt rekkefølge

Input	H	æ	l	l	æ
Operasjon	push	push	push	push	push
Stack					æ
				l	l
			l	l	l
		æ	æ	æ	æ
	H	H	H	H	H
Output	æ	l	l	æ	H
Operasjon	pop	pop	pop	pop	pop
Stack	l				
	l	l			
	æ	æ	æ		
	H	H	H	H	

Implementasjon: Omvendt rekkefølge

- Programmet bruker lærebokas modul for stack implementert som generisk array
- Stackmodulen legges i en egen katalog og inkluderes i Java-programmet ved å bruke en `import`-setning
- `CLASSPATH` må evt. settes riktig slik at Java-kompilatoren finner all koden som brukes
- Java-kode

Anvendelse: Parantessjekking

- Kompilatorer bruker stack bl.a. til å kontrollere korrekt bruk av *paranteser* i programkoden
- Sjekker at parantesene balanserer:
  - For hver `'('` må det finnes en `')'`, for hver `'{'` må det finnes en `'}'`, etc.
  - Antall venstreparanteser må være likt antall høyreparanteser
  - Parantesene må være riktig plassert i forhold til hverandre
- Eksempel:
  - Sekvensen `[ ( ) ]` er riktig, mens sekvensen `[ ( ] )` er feil

Korrekte og feil parantesuttrykk

- Korrekt input:

```
(ab[c]{d[e]})
{(4*a*{b+c}/[d+e])+{f*g}(4)}
```

- Feil input:

```
{a]}
{abc()
(]{}
```

Mulige feil i parantessjekking

1. Mangler venstreparantes: `abc)`
2. Mangler høyreparantes: `(abc`
3. Feil gruppering: `{abc]`

Algoritme for parantessjekking

1. Lag en tom stakk
2. Les ett og ett tegn inntil slutt på input

   2.1 Hvis lest tegn er en venstreparantes, push det på stakken

   2.2 Ellers, hvis lest tegn er en høyreparantes:

      2.2.1 Hvis tom stakk: **Feil**, mangler en venstreparantes

      2.2.2 Ellers, pop stakken og sjekk om dette er en matchende venstreparantes
      Hvis ikke: **Feil** gruppering av paranteser

3. Hvis stakken ikke er tom når alle tegn er lest: **Feil**, mangler minst én høyreparantes

Eksempler: Parantessjekking

Input	(	a	)	]			
Operasjon	push		pop	*feil, mangler venstre*			
Stack	(	(					
Input	(	{	a	]			
Operasjon	push	push		pop – *feil gruppering*			
Stack		{	{				
	(	(	(	(			
Input	(	[	{	a	}	]	EOF
Operasjon	push	push	push		pop	pop	*feil, mangler høyre*
Stack			{	{			
		[	[	[	[		
	(	(	(	(	(	(	(
Input	(	[	]	{	a	}	)	EOF
Operasjon	push	push	pop	push		pop	pop	*0K*
Stack		[		{	{			
	(	(	(	(	(	(		

Implementasjon: Parantessjekking

- Bruker Javas egen modul for stack implementert som en Collection i pakken java.util
- Sjekker bare én linje med parantesuttrykk
- Skriver bare ut `true` hvis korrekt uttrykk, skriver ut `false` hvis feil
- Tegn som regnes som paranteser:

```
( ) { } [ ] < >
```

- Java-kode

Anvendelse av stack: En postfix-maskin

- Vi er vant til å skrive regneuttrykk med **infix** notasjon, der operatorer settes *mellom* operandene:

  a / b

  a + b * c – d

  (a + b) * (c – d)

- Infix krever *presedens*regler og paranteser for å kunne beregnes riktig:
  - Beregnes fra venstre mot høyre
  - Uttrykk inne i paranteser beregnes først
  - Multiplikasjon og divisjon utføres før addisjon og subtraksjon

Postfix notasjon

- I **postfix** notasjon skrives en operator rett *etter de to* operandene som den skal virke på:

Infix	**Postfix**
a / b	a b /
a + b * c – d	a b c * + d –
(a + b) * (c – d)	a b + c d – *

- Postfix trenger ingen presedensregler eller paranteser
- Enkelt å beregne *maskinelt* med bruk av en stack
- “Steinalderprogrammering” brukte postfix

Noen eksempler med heltall

Infix	Postfix
1 + 9	1 9 +
1 + 4 * 2	1 4 2 * +
1 + 2 * 3 + 4 * 5	1 2 3 * + 4 5 * +

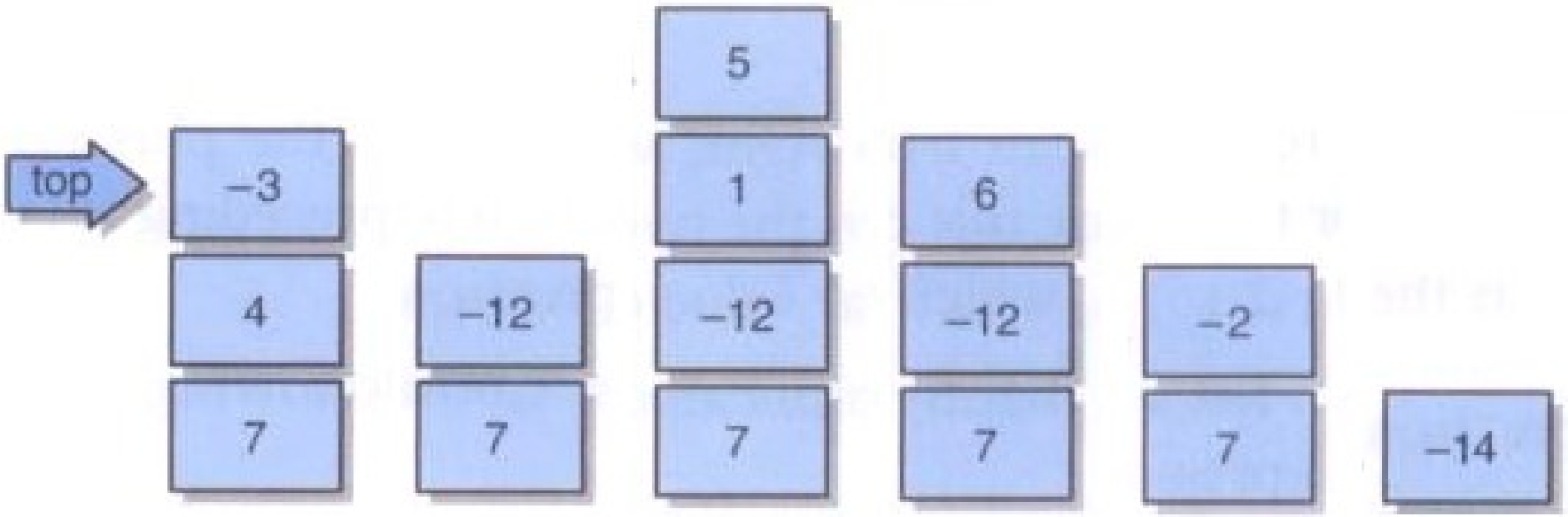
Beregning av postfix regneuttrykk

- Bruker en stack til å beregne verdien av regneuttrykk skrevet (korrekt) i postfix:
  - Les gjennom regneuttrykket fra venstre mot høyre
  - Når en operand leses, pushes denne på stacken
  - Når en operator leses, pop de to øverste elementene på stacken, utfør regneoperasjonen på disse to, og legg resultatet tilbake på stacken
  - Når hele regnestykket er lest ligger verdien alene på stacken

Eksempel: Postfix-beregning

Endringene i stacken ved beregning av:

7 4 −3 * 1 5 + / *



Eksempler: Postfixberegning

Input	**2**	**3**	**+**	**5**	*****					
Stack		3		5						
	2	2	5	5	25					
Input	**1**	**2**	**3**	*****	**+**	**4**	**5**	*****	**+**	
Stack			3				5			
		2	2	6		4	4	20		
	1	1	1	1	7	7	7	7	27	
Input[1]	**a**	**b**	**c**	**-**		*****		**d**		**/**
Stack			c							
		b	b	b-c				d		
	a	a	a	a		a*(b-c)		a*(b-c)		a*(b-c)/d

1: I dette eksemplet vil beregningen oversette regneuttykket fra postfix til infix

Implementasjon: Postfix-beregning

- Begrensninger:
  - Regner bare med positive heltall
  - Bare fire operasjoner: + - * /
- Bruker en egen enkel stack som bare kan lagre heltall *
- Beregning av postfix: Javakode

*: Merk at læreboken har en annen implementasjon av postfix-beregning som bruker den innebygde stack-klassen i Java Collections API (java.util.Stack)

Oversettelse fra infix til postfix

- Mennesker forstår best infix, maskinelle beregninger krever regneuttrykk skrevet i postfix (maskinkode)
- Viktig problem å løse:
  - Oversettelse (kompilering) av infix til postfix – grunnleggende for å kunne lage gode *høynivå* programmeringsspråk
- FORTRAN – FORmula TRANslator (IBM, 1957)
  - Første kommersielle programmeringsspråk som tillot skriving av regneuttrykk i infix
  - Grunnlaget for fremveksten av teknisk programvareindustri

Metode for infix → postfix

- Leser infix-uttrykk fra venstre mot høyre
- Alle operander som leses skrives direkte til postfix-uttrykk
- Operatorene lagres unna på stack, skrives først ut til postfix rett etter at *begge* operandene er skrevet ut
- Høyre operand til operatoren på toppen av stacken er ferdig skrevet ut når det leses en operator fra infix-uttrykket som har *lavere eller lik* prioritet

Forenklinger: Infix → postfix

- Ser bare på forenklede regneuttrykk, for å få en mindre komplisert algoritme uten flere "fiklete" spesialtilfeller

- Forenklinger:
  - Ingen paranteser i infix-uttrykket
  - Alle operander er bare enkle tegn
  - Bare fire operatorer: + - * /
  - Ikke *unært* minus (negative tall)

Algoritme: Infix → postfix

1. Opprett en tom stack *S* som kan lagre operatorer
2. For alle tegn *T* i infix-uttrykket:
   2.1. Hvis *T* er en operand: Skriv ut *T*
   2.2. Ellers, hvis *T* er en operator:
      2.2.1. Så lenge *prioritet(T) ≤ prioritet(S.peek())*
         2.2.1.1 Skriv ut *S.pop()*
      2.2.2. *S.push(T)*
3. Skriv ut alle gjenværende operatorer på *S*

- Implementasjon i Java er gitt som øvingsoppgave

Eksempel: Infix → postfix

- Input: **a*b+c-d/e**

Inn	a	*	b	+	c	-	d	/	e	EOF
Stack		*	*	+	+	-	-	/ -	/ -	
Ut	a		b	*	c	+	d		e	/-

- Output: **ab*c+de/-**

Anvendelse av stack: Labyrint

- 2D-labyrint med gitt antall rader og kolonner:
  - Tabell med verdiene 1 (åpen vei) og 0 (hindring)
  - Starter i øvre venstre hjørne, prøver å finne vei til nedre høyre
  - Kan bare ta steg til venstre, høyre, opp og ned, ikke diagonalt

```
9 13
1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 0 1 1 0 1 1 1 1 0 0 1
1 1 1 1 1 0 1 0 1 0 1 0 0
0 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1
```

Strategi for å finne en vei i labyrinten

- Prøver etter tur alle *mulige* veier videre fra *nåværende* posisjon i labyrinten
- Følger alltid en mulig vei så langt som det går
- Hvis vi kommer til en blindvei:
  - Returnererer til foregående posisjon, i motsatt rekkefølge av veien vi kom (backtracking)
  - Prøver alle de andre *uprøvde* stegene videre derfra
- Besøkte posisjoner merkes, slik at vi ikke går i ring

Løsning med en stack

- Bruker en stack til i hvert steg å lagre alle **lovlige steg** videre fra nåværende posisjon
- Lovlig steg:
  - Går ikke ut av labyrinten
  - Går ikke til en blokkert rute
  - Går ikke til en rute vi har oppsøkt eller lagret fra før
- Starter med å pushe (0, 0) på stacken
- Går i hvert steg videre til øverste posisjon på stacken, inntil vi er fremme i nedre høyre hjørne eller stacken er tom for lovlige steg videre (finnes ingen vei gjennom labyrinten)

4 x 4 Eksempel

	0	1	2	3
0	1	1	1	1
1	0	1	1	0
2	1	1	0	1
3	0	1	1	1

Rekkefølge for push:

	1	
2	X	4
	3	

Pos.		0,0	0,1	0,2	0,3	1,2	1,1	2,1	3,1	3,1	3,2	3,3
Stack	0,0	0,1	0,2 1,1	0,3 1,2 1,1	1,2 1,1	1,1	2,1	3,1 2,0	2,0	3,2 2,0	3,3 2,0	2,0

Representasjon av labyrint i Java

```
int[][] labyrint = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                     {1,0,0,1,1,0,1,1,1,1,0,0,1},
                     {1,1,1,1,1,0,1,0,1,0,1,0,0},
                     {0,0,0,0,1,1,1,0,1,0,1,1,1},
                     {1,1,1,0,1,1,1,0,1,0,1,1,1},
                     {1,0,1,0,0,0,0,1,1,1,0,0,1},
                     {1,0,1,1,1,1,1,1,1,0,1,1,1,1},
                     {1,0,0,0,0,0,0,0,0,0,0,0,0,0},
                     {1,1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

Enkel løsning i Java

- Leser labyrinten inn fra en datafil
- Labyrinten representeres med en 2D-tabell med verdiene:

  0: Ledig 1: Stengt 2: Lagret på stack 3: Oppsøkt

- Bruker stack-klassen fra Java Collections
- Indre klasse `posisjon` for å lagre lovlige steg på stacken
- Sjekker bare om det finnes en vei, tar ikke vare på veien
- Kode: labyrintStack.java

Løsning fra læreboka

- Egen klasse `Position` for å lagre lovlige steg på stacken
- Labyrinten representeres med en klasse `Maze` :
  - Innholdet i labyrinten leses inn fra en datafil
  - Markerer posisjoner som er oppsøkt med verdien 2
  - Har en metode `validPosition` som sjekker lovlig steg
- Finner vei gjennom labyrinten: `MazeSolver`
  - Bruker en innebygget stackstruktur fra Java Collections
  - Finner bare ut *om* det er en vei i labyrinten eller ikke
  - Lagrer ikke veien som er funnet
- Testprogram: `MazeTester`