Implementatie Heap

Inleiding

- Geheugen variabelen e.d. staan op stack
  - wordt automatische opgeruimd na afloop functie call (opschuiven stackpointer)
- Geheugen instanties klassen en arrays niet op stack
  - wordt apart beheerd (op heap)
- Geheugen op heap is niet gebonden aan enkele functie
  - blijft na functie aanroep bestaan.
- Geheugen op heap
  - expliciet gealloceerd
  - impliciet opgeruimd (in Python en Java)

Memory allocatie

Python

- `a = [0] * 100`
  op heap array met lengte 100 gealloceerd: 100 * 4 bytes (8 bytes voor 64 bits architectuur)

a verwijst (is pointer) naar array

b = a[23];

er wordt 23 bij start adres (a) opgeteld en inhoud wordt terugggegeven

Memory Management

Eenvoudige implementatie heap management
- expliciet allocatie
- automatische deallocatie

Heap
- aaneengesloten geheugen gebied (array)
- Essential Python: alleen integers

Administratie
- adres (pointer) naar begin vrij deel heap
- begint bij 0
- iedere aanvraag schuift deze pointer op

Instructies Memory Management

new

- op stack: size van te creeren array
- resultaat: alloceer blokgrootte size, adres op stack

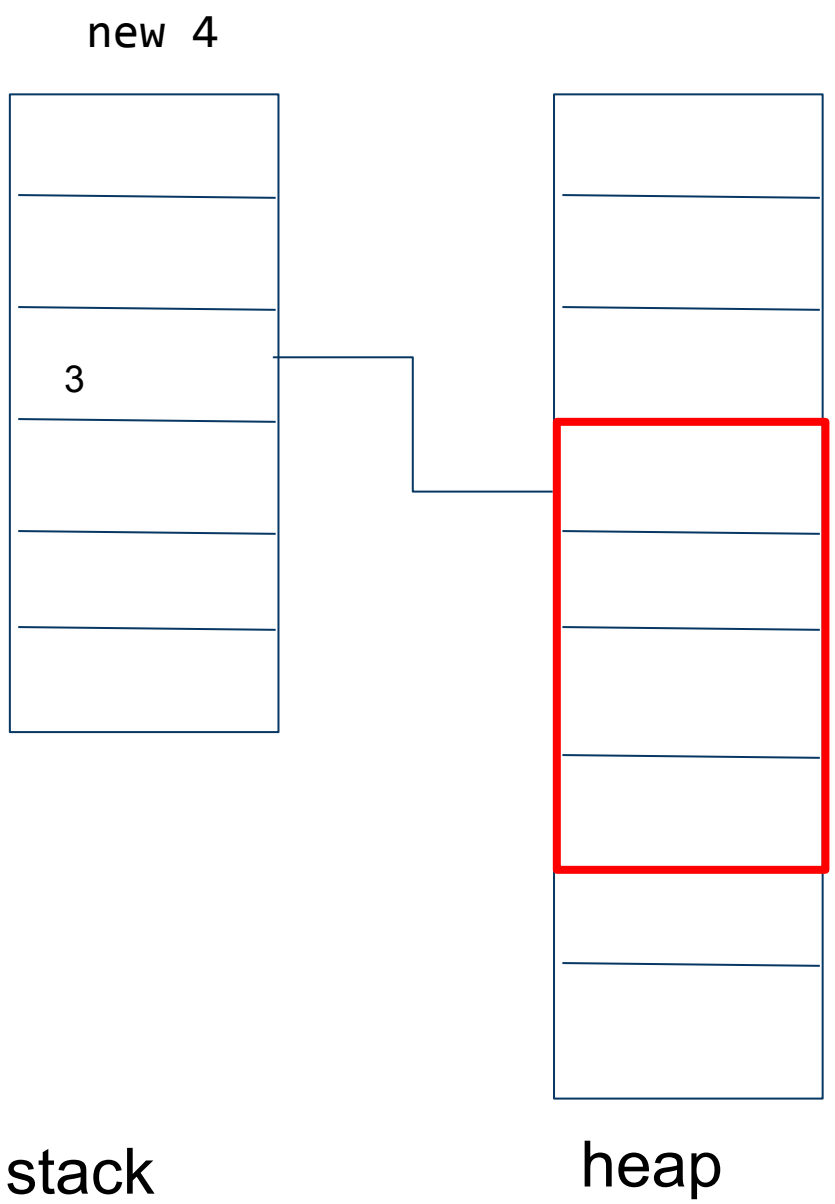
iaload

- op stack: index en adress array
- resultaat: waarde index op stack

iastore

- op stack waarde, index en adres, resultaat waarde op positie index gezet

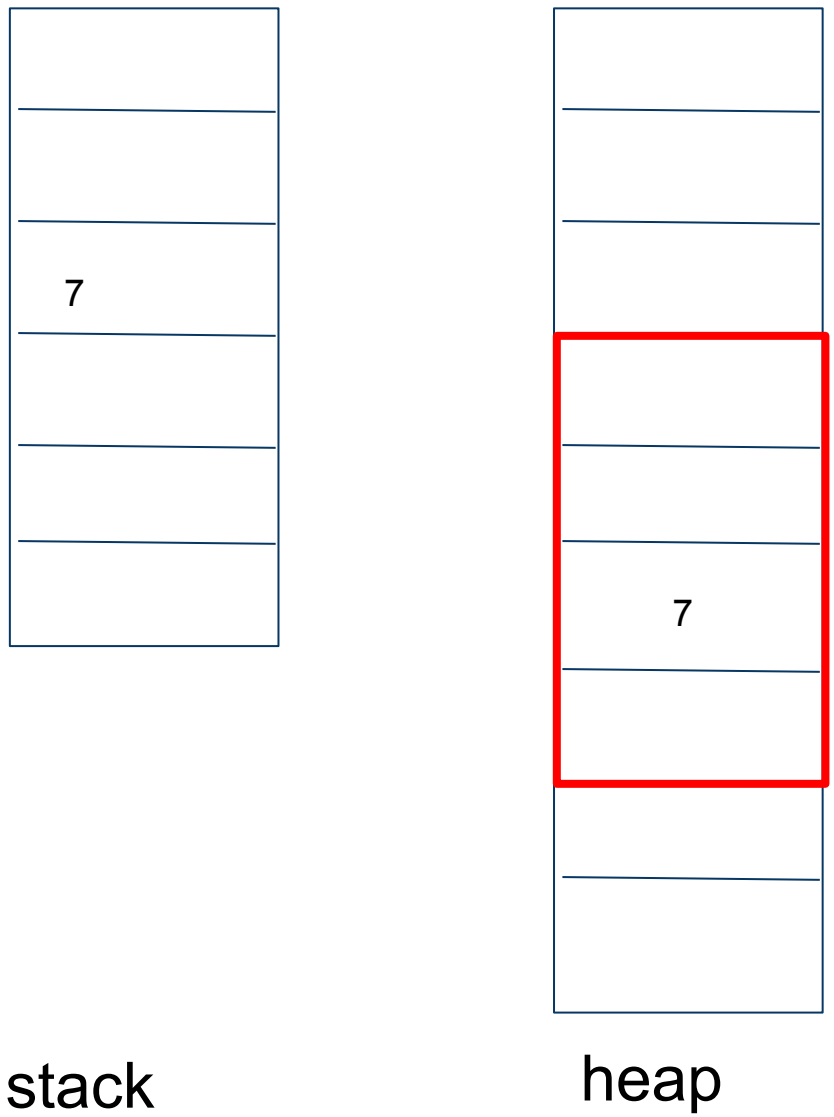
Heap & Stack



new n creeert blok van lengte n op de heap en zet het adres (index) naar het eerste element op de stack
afhankelijk van de implementatie wordt ook n nog ergens opgeslagen op de heap (bv in een extra veld op de heap)!

Heap & Stack

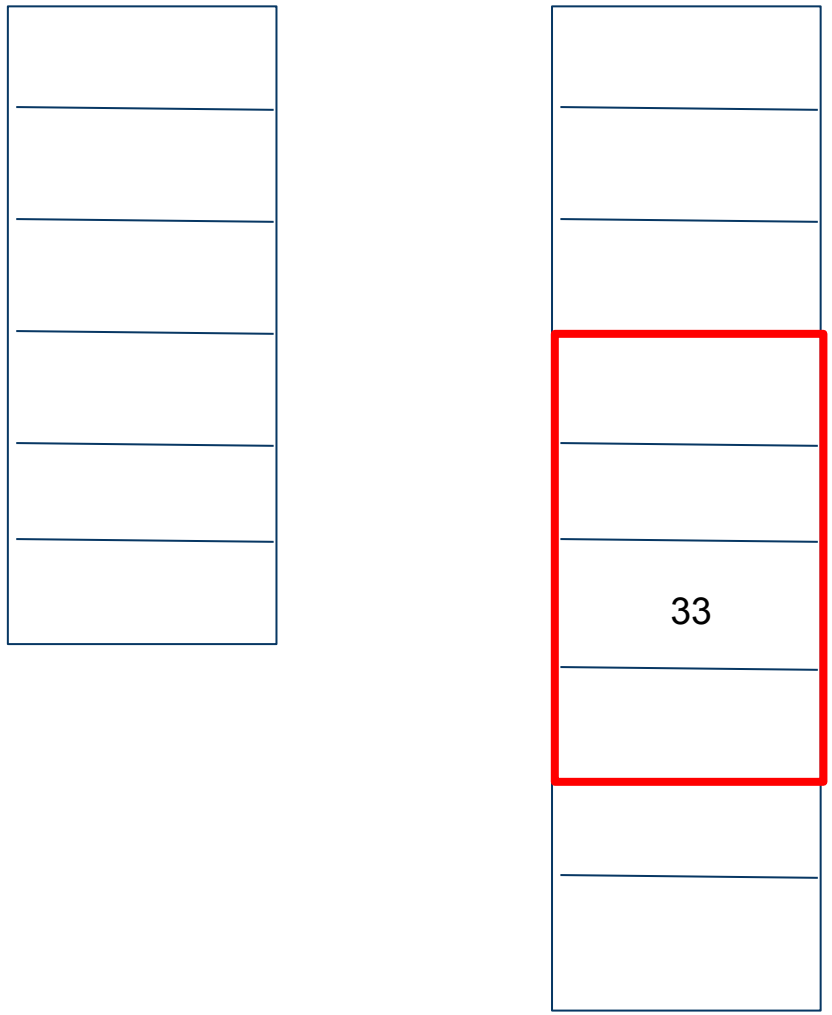
```
push 2
load 0
iaload
```



in iaload wordt de index opgeteld bij de offset en dat element uit de heap wordt op de stack gezet

Heap & Stack

```
push 33
push 2
load 0
iastore
```



in iastore gebeurt het omgekeerde
een waarde op de stack wordt naar de heap gekopieerd

Voorbeelden gebruik

```
a       = [0]*10
```

```
bipush 10
new
istore 0
```

```
a[3]    = 12
```

```
bipush 12
bipush 3
iload 0
iastore
```

```
b       = a[3]
```

```
bipush 3
iload 0
iaload
istore 1
```

Stack & Heap

Let op: aan een waarde op de stack is niet te zien of dit gewoon een getal of een index naar een array (pointer) op de heap is: beide gewoon getallen.

```
push 3
push 3
iaload
```

geeft gewoon het 7e (tel vanaf 0) element van de heap!

In Python kan dit niet gebeuren, want alle array operaties worden gecontroleerd (door extra instructies te genereren)!

Stack & Heap

Moderne programmeertalen beschermen de toegang tot de heap:

- iedere heap verwijzing op de stack is gemarkeerd en dus herkenbaar (extra bit oid)
- ieder heap blok bevat administratie over grootte e.d
- ieder access wordt gecontroleerd

In C(++) gebeurt dat niet en ben je dus in staat op willekeurige plekken in het geheugen een waarde te lezen of te schrijven.

Voorbeeld Selection Sort

```
def test():
 a = [0] * 5
 n = 5
 a[0] = 6
 a[1] = 9
 a[2] = 3
 a[3] = 7
 a[4] = 2
 print(selsort(a,5))
```

```
def selsort(a,n):
    i = 0
    while i < n:
      m = i
      j = i + 1
      while j < n:
        if a[j] < a[m]:
          m = j
        j = j + 1
      h = a[i]
      a[i] = a[m]
      a[m] = h
      i = i + 1
    return a
```

Voorbeeld Selection Sort

```
    call 0 2 5
    pop
    stop
5   bipush 5
    new
    istore 0
    bipush 5
    istore 1
    bipush 6
    bipush 0
    iload 0
    iastore
    bipush 9
    bipush 1
    iload 0
    iastore
    bipush 3
    bipush 2
    iload 0
    iastore
    bipush 7
    bipush 3
    iload 0
    iastore
    bipush 2
    bipush 4
    iload 0
    iastore
    iload 0
    printmem
    iload 0
    iload 1
    call 2 4 10
    printmem
    bipush 77
    ireturn
10  bipush 0
    istore 2
20  iload 2
    iload 1
    isub
    ifeq 50
    iload 2
    istore 4
    iload 2
    bipush 1
    iadd
    istore 3
30  iload 3
    iload 1
    isub
    ifeq 40
    iload 0
    iload 3
        iaload
        iload 0
        iload 4
        iaload
        isub
        iflt 45
        goto 47
45      iload 3
        istore 4
47      iinc 3 1
        goto 30
40      iload 0
        iload 2
        iaload
        istore 5
        iload 0
        iload 4
        iaload
        iload 0
        iload 2
        iastore
        iload 5
        iload 0
        iload 4
        iastore
        iinc 2 1
        goto 20
50      iload 0
        ireturn
```

Automatische garbage collection

Python

- vrijgeven automatisch.
- aanvragen zoals hierboven
- als hoeveelheid vrij geheugen beneden kritische grens
  - stop programma en start garbage collection
    - loop stack af
    - markeer gealloceerd geheugen vanaf stack (direct en indirect!) bereikbaar
    - overgebleven gealloceerd geheugen wordt vrijgegeven
  - voorwaarde
    - verschil adres en integer stack variabelen moet zichtbaar zijn (als extra bit oid)