AULA 2

Análise amortizada

Fonte: https://www.europosters.pt/telas/

CLRS 17

Contador binário

Incrementa de 1 o número binário representado por $a[0\,..\,k-1]$.

INCREMENT (a, k)
1 $i \leftarrow 0$
2 **enquanto** $i < k$ **e** $a[i] = 1$ **faça**
3 $\quad a[i] \leftarrow 0$
4 $\quad i \leftarrow i + 1$
5 **se** $i < k$
6 $\quad$ **então** $a[i] \leftarrow 1$

Contador binário

Incrementa de 1 o número binário representado por $a[0\,..\,k-1]$.

INCREMENT (a, k)
1 $i \leftarrow 0$
2 **enquanto** $i < k$ **e** $a[i] = 1$ **faça**
3 $\quad a[i] \leftarrow 0$
4 $\quad i \leftarrow i + 1$
5 **se** $i < k$
6 $\quad$ **então** $a[i] \leftarrow 1$

Entrada:

$k-1$		3	2	1	0	
0	1	0	1	1	1	a

Contador binário

Incrementa de 1 o número binário representado por $a[0\,..\,k-1]$.

INCREMENT (a, k)
1 $i \leftarrow 0$
2 **enquanto** $i < k$ **e** $a[i] = 1$ **faça**
3 $\quad a[i] \leftarrow 0$
4 $\quad i \leftarrow i + 1$
5 **se** $i < k$
6 $\quad$ **então** $a[i] \leftarrow 1$

Entrada:

$k-1$		3	2	1	0	
0	1	0	1	1	1	a

Saída:

$k-1$		3	2	1	0	
0	1	1	0	0	0	a

Consumo de tempo

linha	consumo de todas as execuções da linha
1	$\Theta(1)$
2	$O(k)$
3	$O(k)$
4	$O(k)$
5	$\Theta(1)$
6	$O(1)$
total	$O(k) + \Theta(1) = O(k)$

"Custo" $=$

consumo de tempo $=$ número de bits alterados $= O(k)$

Sequência de n chamadas

a começa zerado.

$$\underbrace{\textsc{Incr}\ \ \textsc{Incr}\ \ \cdots\ \ \textsc{Incr}\ \ \textsc{Incr}\ \ \textsc{Incr}}_{n}$$

Consumo de tempo é $O(nk)$

Sequência de n chamadas

a começa zerado.

$$\underbrace{\textsc{Incr}\ \ \textsc{Incr}\ \ \cdots\ \ \textsc{Incr}\ \ \textsc{Incr}\ \ \textsc{Incr}}_{n}$$

Consumo de tempo é $O(nk)$

EXAGERO!

Exemplo

$n = 16$ $k = 6$

a 5	4	3	2	1	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1

a 5	4	3	2	1	0
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0

Exemplo

$n = 16$ $k = 6$

a 5	4	3	2	1	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1

a 5	4	3	2	1	0
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0

$a[0]$	muda	n	vezes

Exemplo

$n = 16$ $k = 6$

a 5	4	3	2	1	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1

a 5	4	3	2	1	0
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0

$a[0]$	muda	n	vezes
$a[1]$	"	$\lfloor n/2 \rfloor$	"

Exemplo

$n = 16$ $k = 6$

a 5	4	3	2	1	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	0	1
0	0	0	1	1	0
0	0	0	1	1	1

a 5	4	3	2	1	0
0	0	1	0	0	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	0	1
0	0	1	1	1	0
0	0	1	1	1	1
0	1	0	0	0	0

$a[0]$	muda	n	vezes
$a[1]$	"	$\lfloor n/2 \rfloor$	"
$a[2]$	"	$\lfloor n/4 \rfloor$	"

Exemplo

$n = 16$ $\quad$ $k = 6$

a	a
5 4 3 2 1 0	5 4 3 2 1 0
0 0 0 0 0 0	0 0 1 0 0 0
0 0 0 0 0 1	0 0 1 0 0 1
0 0 0 0 1 0	0 0 1 0 1 0
0 0 0 0 1 1	0 0 1 0 1 1
0 0 0 1 0 0	0 0 1 1 0 0
0 0 0 1 0 1	0 0 1 1 0 1
0 0 0 1 1 0	0 0 1 1 1 0
0 0 0 1 1 1	0 0 1 1 1 1
	0 1 0 0 0 0

a[0]	muda	n	vezes
a[1]	"	$\lfloor n/2 \rfloor$	"
a[2]	"	$\lfloor n/4 \rfloor$	"
a[3]	"	$\lfloor n/8 \rfloor$	"

Análise agregada

Custo total:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n = \Theta(n)$$

Custo amortizado (= custo médio) de uma operação:

$$\frac{2n}{n} = \Theta(1)$$

Este foi o **método agregado** de análise: soma os custos de todas as operações para determinar o custo amortizado de cada operação

Custo amortizado

O **custo amortizado** de uma operação é o custo médio da operação quando considerada em uma sequência de operações do ADT.

Conclusões

> O consumo de tempo de uma sequência de n execuções do algoritmo INCREMENT é $\Theta(n)$.

> O consumo de tempo amortizado do algoritmo INCREMENT é $\Theta(1)$.

Método de análise contábil

a começa zerado.

Pague $2 para mudar a[i] de $0 \to 1$

$0 para mudar a[i] de $1 \to 0$

Método de análise contábil

a começa zerado.

Pague $2 para mudar a[i] de $0 \to 1$

$0 para mudar a[i] de $1 \to 0$

a[i] muda de $0 \to 1$ $\begin{cases} \$1 \text{ é pago pela operação} \\ \$1 \text{ é colocado na poupança.} \end{cases}$

a[i] muda de $1 \to 0$:
paga com poupança do i-ésimo bit.

Método de análise contábil

a começa zerado.

Pague $2 para mudar a[i] de $0 \to 1$

$0 para mudar a[i] de $1 \to 0$

a[i] muda de $0 \to 1$ $\begin{cases} \$1 \text{ é pago pela operação} \\ \$1 \text{ é colocado na poupança.} \end{cases}$

a[i] muda de $1 \to 0$:
paga com poupança do i-ésimo bit.

Custo amortizado por chamada de INCREMENT: $\leq$ $2 (no máximo uma mudança $0 \to 1$ é feita).

Método de análise contábil

a começa zerado.

Pague $2 para mudar a[i] de $0 \to 1$

$0 para mudar a[i] de $1 \to 0$

a[i] muda de $0 \to 1$ $\begin{cases} \$1 \text{ é pago pela operação} \\ \$1 \text{ é colocado na poupança.} \end{cases}$

a[i] muda de $1 \to 0$:
paga com poupança do i-ésimo bit.

Custo amortizado por chamada de INCREMENT: $\leq$ $2 (no máximo uma mudança $0 \to 1$ é feita).

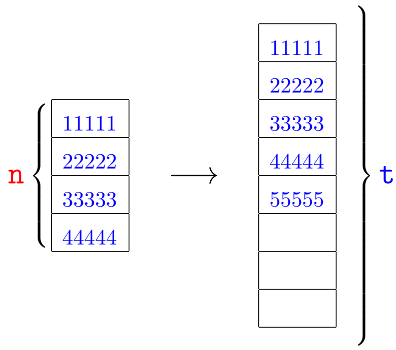
Como $ armazenado nunca é negativo, uma sequência de n chamadas de INCREMENT custa

Tabelas dinâmicas

Fonte: https://twitter.com/MinionPostDoc

CLRS 17

Tabelas dinâmicas



n[T]= número de itens   t[T]= tamanho de T

Inicialmente $n[T] = t[T] = 0$

Inserção

TABLE-INSERT (T, x) ▷ Insere x na tabela T

1 **se** $t[T] = 0$
2 **então** aloque *tabela*[T] com 1 posição
3 $t[T] \leftarrow 1$
4 **se** $n[T] = t[T]$
5 **então** aloque *nova-tabela* com $2\,t[T]$ pos.
6 insira itens da *tabela*[T] na *nova-tabela*
7 $t[\textit{nova-tabela}] \leftarrow 2\,t[T]$
8 libere *tabela*[T]
9 *tabela*[T] $\leftarrow$ *nova-tabela*
10 insira x na *tabela*[T]
11 $n[T] \leftarrow n[T] + 1$

Custo = número de inserções elementares (linhas 6 e 10)

Sequência de m TABLE-INSERTS

$$T_0 \xrightarrow{1^{\text{a}}\text{ op}} T_1 \xrightarrow{2^{\text{a}}\text{ op}} T_2 \longrightarrow \cdots \xrightarrow{m^{\text{a}}\text{ op}} T_m$$

T_i = estado de T depois da i^{a} operação.

Sequência de $\mathtt{m}$ TABLE-INSERTs

$$\mathtt{T}_0 \xrightarrow{1^{\text{a}}\text{ op}} \mathtt{T}_1 \xrightarrow{2^{\text{a}}\text{ op}} \mathtt{T}_2 \longrightarrow \cdots \xrightarrow{\mathtt{m}^{\text{a}}\text{ op}} \mathtt{T}_{\mathtt{m}}$$

$\mathtt{T}_i$ = estado de $\mathtt{T}$ depois da i^{a} operação.

Custo real da i^{a} operação:

$$c_i = \begin{cases} 1 & \text{se há espaço} \\ \mathtt{n}_i & \text{se tabela cheia,} \end{cases}$$

$$\begin{aligned} \text{onde } \mathtt{n}_i &= \text{valor de } \mathtt{n}[\mathtt{T}] \text{ depois da } i^{\text{a}} \text{ operação} \\ &= i. \end{aligned}$$

Sequência de $\mathtt{m}$ TABLE-INSERTs

$$\mathtt{T}_0 \xrightarrow{1^{\text{a}}\text{ op}} \mathtt{T}_1 \xrightarrow{2^{\text{a}}\text{ op}} \mathtt{T}_2 \longrightarrow \cdots \xrightarrow{\mathtt{m}^{\text{a}}\text{ op}} \mathtt{T}_{\mathtt{m}}$$

$\mathtt{T}_i$ = estado de $\mathtt{T}$ depois da i^{a} operação.

Custo real da i^{a} operação:

$$c_i = \begin{cases} 1 & \text{se há espaço} \\ \mathtt{n}_i & \text{se tabela cheia,} \end{cases}$$

$$\begin{aligned} \text{onde } \mathtt{n}_i &= \text{valor de } \mathtt{n}[\mathtt{T}] \text{ depois da } i^{\text{a}} \text{ operação} \\ &= i. \end{aligned}$$

Custo de uma operação = $\mathrm{O}(\mathtt{m})$.

Sequência de $\mathtt{m}$ TABLE-INSERTs

$$\mathtt{T}_0 \xrightarrow{1^{\text{a}}\text{ op}} \mathtt{T}_1 \xrightarrow{2^{\text{a}}\text{ op}} \mathtt{T}_2 \longrightarrow \cdots \xrightarrow{\mathtt{m}^{\text{a}}\text{ op}} \mathtt{T}_{\mathtt{m}}$$

$\mathtt{T}_i$ = estado de $\mathtt{T}$ depois da i^{a} operação.

Custo real da i^{a} operação:

$$c_i = \begin{cases} 1 & \text{se há espaço} \\ \mathtt{n}_i & \text{se tabela cheia,} \end{cases}$$

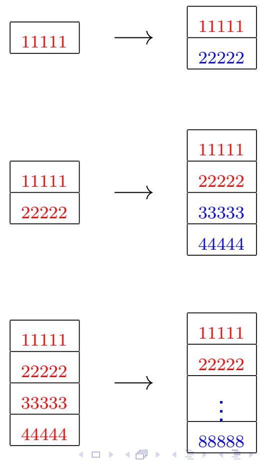
$$\begin{aligned} \text{onde } \mathtt{n}_i &= \text{valor de } \mathtt{n}[\mathtt{T}] \text{ depois da } i^{\text{a}} \text{ operação} \\ &= i. \end{aligned}$$

Custo de uma operação = $\mathrm{O}(\mathtt{m})$.

Custo das $\mathtt{m}$ operações = $\mathrm{O}(\mathtt{m}^2)$. Exagero!

Exemplo

$\mathtt{n}[\mathtt{T}]$ (operação)	$\mathtt{t}[\mathtt{T}]$	custo
1	1	1
2	2	$1+1$
3	4	$1+2$
4	4	1
5	8	$1+4$
6	8	1
7	8	1
8	8	1
9	16	$1+8$
10	16	1
16	16	1
17	32	$1+16$
33	64	$1+32$



Custo amortizado

Custo total:

$$\sum_{i=1}^{\mathtt{m}} c_i \;=\; \mathtt{m} + \sum_{i=0}^{k} 2^i \;=\; \mathtt{m} + 2^{k+1} - 1 \;<\; \mathtt{m} + 2\mathtt{m} - 1 \;<\; 3\mathtt{m}$$

onde $k = \lfloor \lg(\mathtt{m} - 1) \rfloor$

Custo amortizado:

$$\frac{3\mathtt{m}}{\mathtt{m}} \;=\; 3 \;=\; \Theta(1)$$

Conclusões

O custo de uma sequência de $\mathtt{m}$ execuções do algoritmo TABLE-INSERT é $\Theta(\mathtt{m})$.

O custo amortizado do algoritmo TABLE-INSERT é $\Theta(1)$.

Método de análise agregada

- $\mathtt{m}$ operações consomem tempo $T(\mathtt{m})$.

Método de análise agregada

- $\mathtt{m}$ operações consomem tempo $T(\mathtt{m})$.
- custo médio de cada operação é $T(\mathtt{m})/\mathtt{m}$.
- custo amortizado de cada operação é $T(\mathtt{m})/\mathtt{m}$.

Método de análise agregada

- $\mathtt{m}$ operações consomem tempo $T(\mathtt{m})$.
- custo médio de cada operação é $T(\mathtt{m})/\mathtt{m}$.
- custo amortizado de cada operação é $T(\mathtt{m})/\mathtt{m}$.
- defeito: no caso de mais de um tipo de operação, o custo de cada tipo não é determinado separadamente.

Método de análise contábil

TABLE-INSERT (T, x)

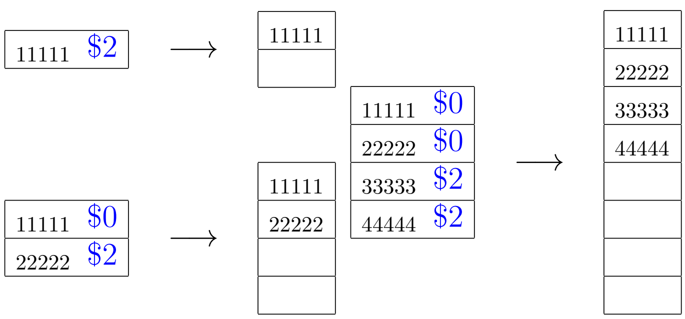
```
    credito ← credito + 3
se t[T] = 0
    então aloque tabela[T] com 1 posição
          t[T] ← 1
se n[T] = t[T]
    então aloque nova-tabela com 2 t[T] pos.
          insira itens da tabela[T] na nova-tabela
          custo ← custo + n[T]
          libere tabela[T]
          tabela[T] ← nova-tabela
          t[T] ← 2 t[T]
insira x na tabela[T]
n[T] ← n[T] + 1
custo ← custo + 1
```

Método de análise contábil

Invariante: soma créditos $\geq$ soma custos reais

n[T]	t[T]	custo	crédito	saldo
1	1	1	3	2
2	2	1+1	3	3
3	4	1+2	3	3
4	4	1	3	5
5	8	1+4	3	3
6	8	1	3	5
7	8	1	3	7
8	8	1	3	9
9	16	1+8	3	3
10	16	1	3	5
16	16	1	3	17
17	32	1+16	3	3

Método de análise contábil



Método de análise contábil

Pague	\$1 para inserir um novo elemento
guarde	\$1 para eventualmente mover o novo elemento
guarde	\$1 para mover um elemento que já está na tabe

Custo amortizado por chamada de TABLE-INSERT:
$\leq$ \$3

Sequência de $\mathtt{m}$ chamadas de TABLE-INSERT.

Como \$ armazenado nunca é negativo,

$$\begin{aligned}\text{soma custos reais} &\leq \text{soma custos amortizados}\\ &= 3\mathtt{m}\\ &= \mathrm{O}(\mathtt{m})\end{aligned}$$

Método de análise contábil

- cada operação paga seu custo real
- cada operação recebe um certo número de créditos
  (chute de custo amortizado)
- balanço nunca pode ser negativo

  $$\text{soma créditos} \geq \text{soma custos reais}$$

  créditos não usados são guardados para pagar operações futuras.
- custo amortizado de cada tipo de operação pode ser determinado separadamente

Conclusões

> O custo de uma sequência de $\mathtt{m}$ execuções do algoritmos TABLE-INSERT é $\Theta(\mathtt{m})$.

> O custo amortizado do algoritmo TABLE-INSERT é $\Theta(1)$.

Sequência de INSERT e DELETE

Sequência de operações TABLE-INSERT e
TABLE-DELETE

$$\underbrace{\mathsf{I\ \ I\ \ D\ \ I\ \ I\ \ D\ \ I\ \ D\ \ D\ \ I\ \ D\ \ D\ \ I\ \ I}}_{\mathtt{m}}$$

Custo total de uma sequência de TABLE-INSERT e TABLE-DELETE?

Remoção

Remove um elemento x da tabela $\mathtt{T}$

TABLE-DELETE $(\mathtt{T}, x)$ ▷ supõe x na *tabela*[T]

```
1  remova x da tabela[T]
2  n[T] ← n[T] − 1
3  se n[T] < t[T]/2     ▷ tabela está "vazia"?
4       então aloque nova-tabela com t[T]/2 pos.
5             insira itens da tabela[T] na nova-tabela
6             t[nova-tabela] ← t[T]/2
7             n[nova-tabela] ← n[T]
8             libere tabela[T]
9             tabela[T] ← nova-tabela
```

Custo = número de remoções e inserções elementares
(linhas 1 e 5)

Sequência de INSERT e DELETE

Sequência de operações TABLE-INSERT e
TABLE-DELETE

$$\underbrace{\underbrace{\mathsf{I}\ \cdots\ \mathsf{I}}_{\mathtt{m}/2}\ \mathsf{I\ \ I\ \ D\ \ D\ \ I\ \ I\ \ D\ \ D\ \ I\ \ I\ \ D\ \ D}}_{\mathtt{m}}$$

Se $\mathtt{m} = 4\,k$, então, o custo total da sequência:

$$\Theta(2\,k) + k\,\Theta(2\,k) = \Theta\Big(\frac{\mathtt{m}}{2}\Big) + \frac{\mathtt{m}}{4}\,\Theta\Big(\frac{\mathtt{m}}{2}\Big) = \Theta(\mathtt{m}^2)$$

Remoção

Remove um elemento x da tabela T

TABLE-DELETE (T, x) ▷ supõe x na *tabela*[T]

```
1  remova x da tabela[T]
2  n[T] ← n[T] − 1
3  se n[T] < t[T]/4     ▷ tabela está "vazia"?
4      então aloque nova-tabela com t[T]/2 pos.
5            insira itens da tabela[T] na nova-tabela
6            t[nova-tabela] ← t[T]/2
7            n[nova-tabela] ← n[T]
8            libere tabela[T]
9            tabela[T] ← nova-tabela
```

Custo = número de remoções e inserções elementares
(linhas 1 e 5)

Sequência de m operações

$$T_0 \xrightarrow{1^a\ op} T_1 \xrightarrow{2^a\ op} T_2 \longrightarrow \cdots \xrightarrow{m^a\ op} T_m$$

T_i = estado de T depois da i^a operação.

Custo real da i^a operação se for TABLE-INSERT:

$$c_i = \begin{cases} 1 & \text{se há espaço} \\ n_i & \text{se tabela cheia} \end{cases}$$

onde n_i = valor de $n[T]$ depois da i^a operação

Custo de uma operação = $O(m)$

Sequência de m operações

$$T_0 \xrightarrow{1^a\ op} T_1 \xrightarrow{2^a\ op} T_2 \longrightarrow \cdots \xrightarrow{m^a\ op} T_m$$

T_i = estado de T depois da i^a operação.

Custo real da i^a operação se for TABLE-DELETE:

$$c_i = \begin{cases} 1 & \text{se } n_{i-1} > t_{i-1}/4 \\ 1 + n_i & \text{se } n_{i-1} = t_{i-1}/4 \end{cases}$$

onde n_i = valor de $n[T]$ depois da i^a operação
e t_i = valor de $t[T]$ depois da i^a operação

Custo de uma operação = $O(m)$

Custo das m operações = $O(m^2)$ Exagero!

Conclusões

O custo de uma sequência de m execuções dos algoritmos TABLE-INSERT e TABLE-DELETE é $\Theta(m)$.

O custo amortizado dos algoritmos TABLE-INSERT e TABLE-DELETE é $\Theta(1)$.

Class ArrayList

https://docs.oracle.com/.../util/ArrayList.html

"... Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost. ..."

Listas em Python

"... CPython's lists are really variable-length arrays, ... The implementation uses a contiguous array of references to other objects, ...

This makes indexing a list `a[i]` *an operation whose cost is independent of the size of the list or the value of the index.*

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve ...; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize."

Veja Design and History FAQ e Laurent Luce's Blog

Pilhas redimensionáveis

Fonte: https://br.pinterest.com/

Pilha (= stack) e sua API (PF)
1.3 Bags, Queues, and Stacks (SW)

Pilhas redimensionáveis

Considere a implementação de saco (Bag) em vetor com redimensionamento.

O custo amortizado da operação add() é muito baixo, pois cada ocorrência de uma execução lenta de add() é precedida por muitas ocorrências de execuções rápidas.

Pilhas redimensionáveis

public class	Stack<Item>	implements iterable<Item>
	Stack()	construtor cria uma pilha de Items vazia
void	push(Item item)	insere item nesta pilha
Item	pop()	remove o Item mais recente desta pilha
boolean	isEmpty()	esta pilha está vazia?
int	size()	número de Items nesta pilh
iterator<Item>	iterator()	iterador de itens

Class Stack: esqueleto

```
import java.util.Iterator;

public class Stack<Item> implements
                      Iterable<Item> {
   private Item[] a; // array of items
   private int n; // number of elements
   public Stack() {...}
   public boolean isEmpty() {...}
   public int size() {...}
   public void push(Item item) {...}
   public Item pop() {...}
   private void resize(int capacity) {...}
   public Iterator<Item> iterator() {...}
}
```

Cliente

```
public static void main(String[] args) {
   Stack<String> stack;
   stack = new Stack<String>();
   while (!StdIn.isEmpty()) {
      String item = StdIn.readString();
      if (!item.equals("-"))
         stack.push(item);
      else if (!stack.isEmpty())
         StdOut.println(stack.pop() + );
   }
   StdOut.println("("+ stack.size() +
                  "left on stack)");
}
```

Stack: isEmpty() e size()

```
// constrói uma pilha vazia
public Stack() {
   a = (Item[]) new Object[2];
   n = 0;
}

public boolean isEmpty() {
   return n == 0;
}

public int size() {
   return n;
}
```

Stack: push() e pop()

```
public void push(Item item) {
  if(n == a.length) resize(2*a.length);
  a[n++] = item; // insere item
}
public Item pop() {
  Item item = a[n-1];
  a[n-1] = null; // evita loitering
  n--;
  // shrink size of array if necessary
  if (n> 0 && n == a.length/4)
     resize(a.length/2);
  return item;
}
```

Stack: resize()

```
private void resize(int capacity) {
  assert capacity >= n;
  // Algorithms implementation
  Item[] t= (Item[])new Object[capacity];
  for(int i = 0; i < n; i++) {
     t[i] = a[i];
  }
  a = tmp;
}
```

Stack: resize()

```
private void resize(int capacity) {
  assert capacity >= n;
  // Algorithms implementation
  Item[] t= (Item[])new Object[capacity];
  for(int i = 0; i < n; i++) {
     t[i] = a[i];
  }
  a = tmp;
}
```

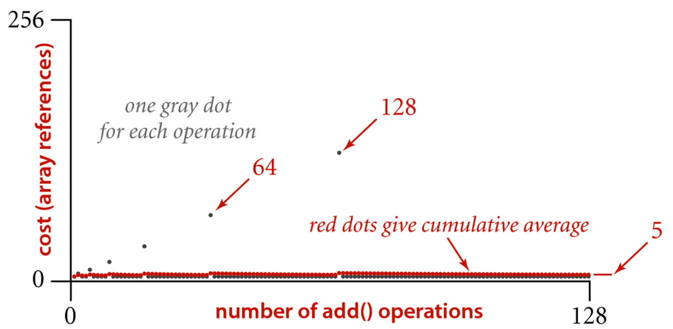
Stack: iterator()

```
public Iterator<Item> iterator() {
  return new ReverseArrayIterator();
}
private class ReverseArrayIterator
        implements Iterator<Item> {
  private int i
  public ReverseArrayIterator() {
     i = n-1;
  }
  [...]
```

Stack: iterator()

```
  public boolean hasNext() {
     return i >= 0;
  }
  public Item next() {
     if (hasNext())
        throw new NoSuchElementException();
     return a[i-];
  }
  public void remove() {
     throw new UnsupportedOperationException();
  }
}
```

Bags redimensionáveis



Amortized cost of adding to a RandomBag