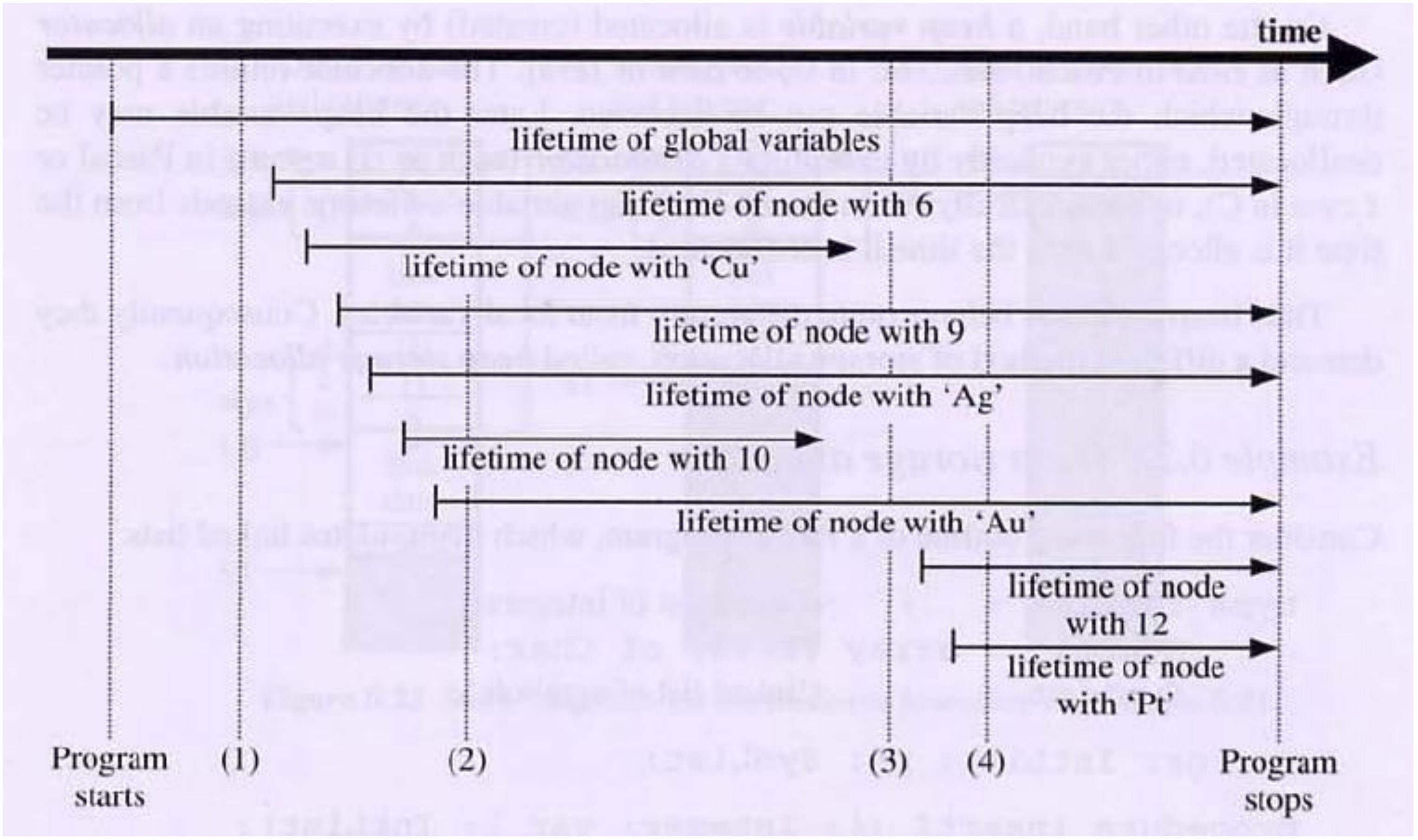
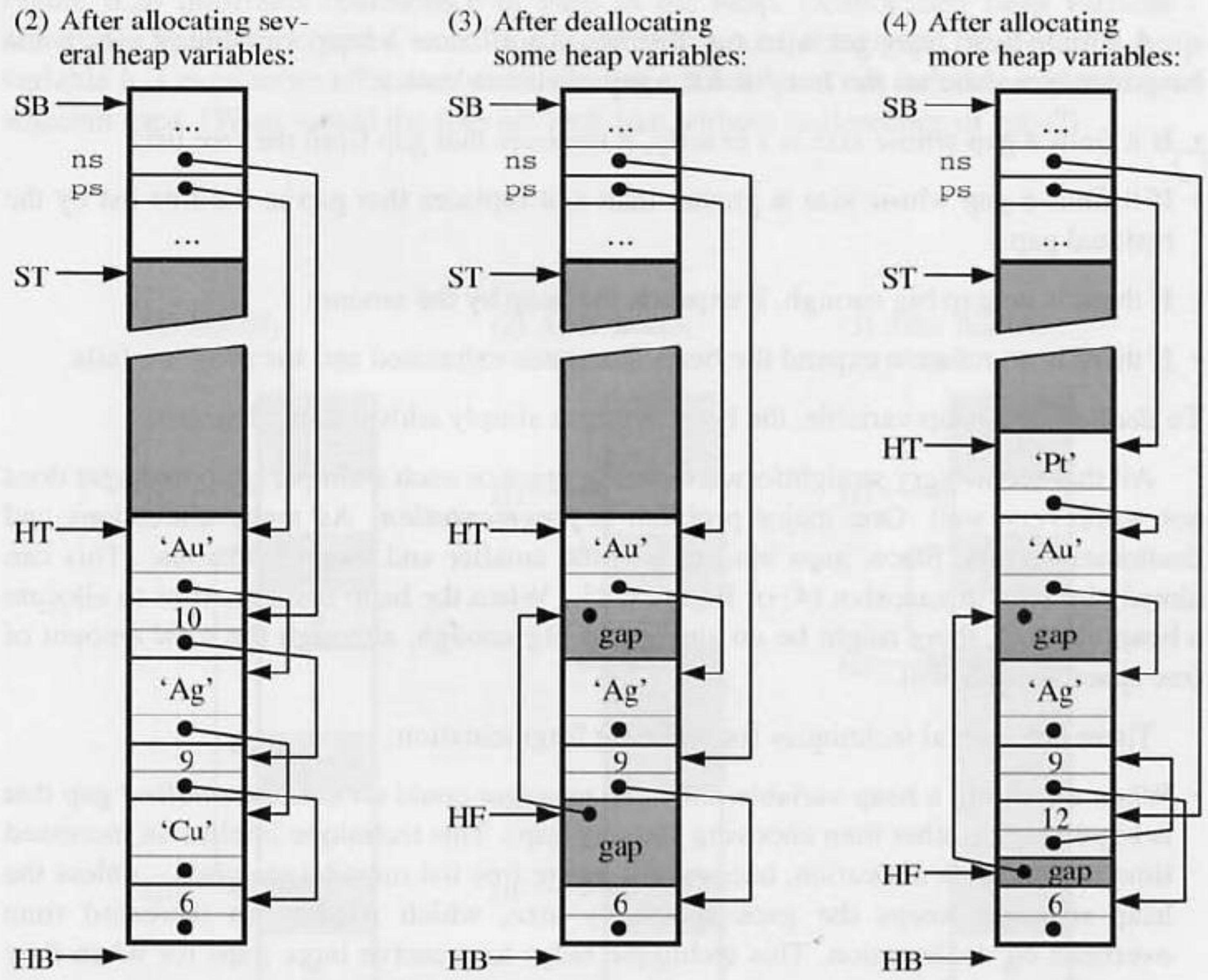
HEAP

Baseado no Capítulo 6 de Programming Language Processors in Java, de Watt & Brown

Última modificação: 10/05/2024 08:25:36

```
type IntList = ...;     {linked list of integers}
     Symbol  = array [1..2] of Char;
     SymList = ...;     {linked list of symbols}

var ns: IntList; ps: SymList;

procedure insertI (i: Integer; var l: IntList);
   ...;     {Insert a node containing i at the front of list l.}

procedure deleteI (i: Integer; var l: IntList);
   ...;     {Delete the first node containing i from list l.}

procedure insertS (s: Symbol; var l: SymList);
   ...;     {Insert a node containing s at the front of list l.}

procedure deleteS (s: Symbol; var l: SymList);
   ...;     {Delete the first node containing s from list l.}

...
ns := nil;          ps := nil;               (1)
insertI(6, ns);     insertS('Cu', ps);
insertI(9, ns);     insertS('Ag', ps);
insertI(10, ns);    insertS('Au', ps);       (2)
deleteI(10, ns);    deleteS('Cu', ps);       (3)
insertI(12, ns);    insertS('Pt', ps);       (4)
```


time
lifetime of global variables
lifetime of node with 6
lifetime of node with 'Cu'
lifetime of node with 9
lifetime of node with 'Ag'
lifetime of node with 10
lifetime of node with 'Au'
lifetime of node with 12
lifetime of node with 'Pt'
Program starts
(1)
(2)
(3)
(4)
Program stops



(2) After allocating several heap variables:
(3) After deallocating some heap variables:
(4) After allocating more heap variables:
SB
ns
ps
...
ST
HT
HF
HB
'Au'
10
'Ag'
9
'Cu'
6
gap
'Pt'
12

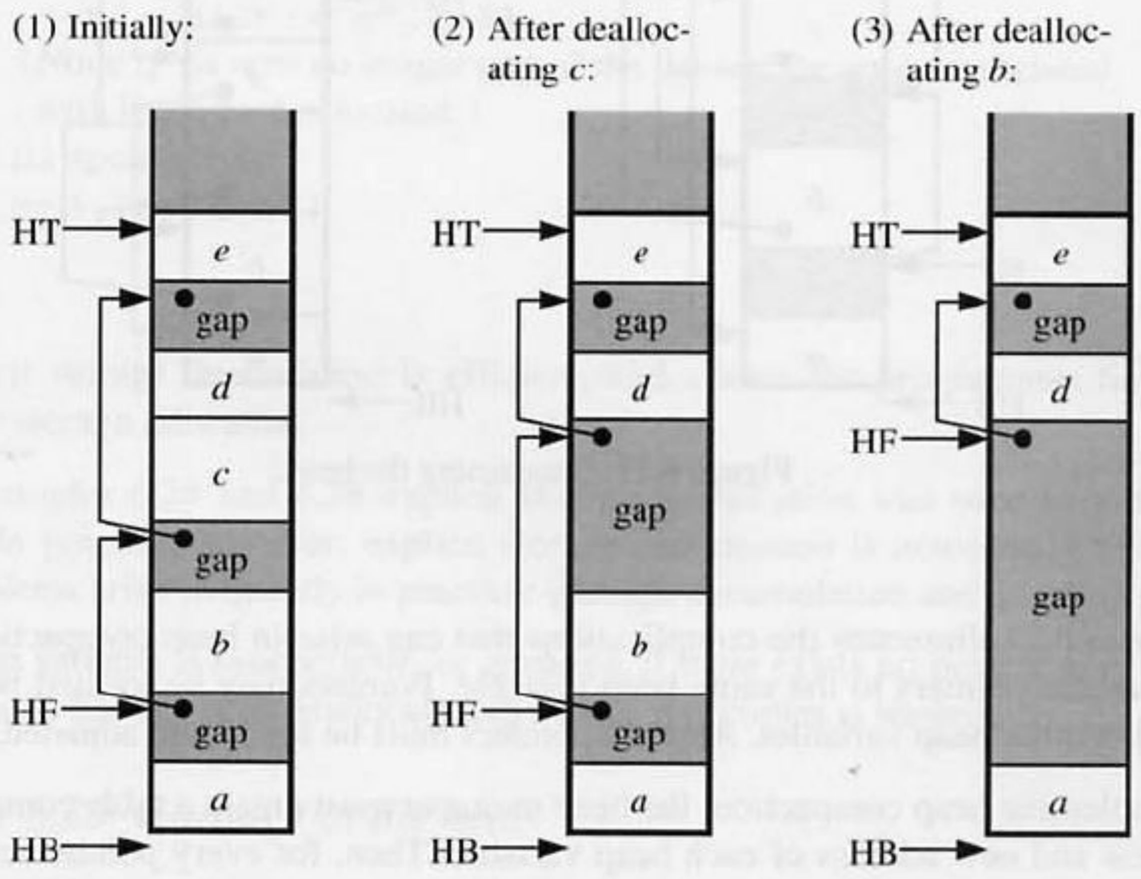
Estratégias:

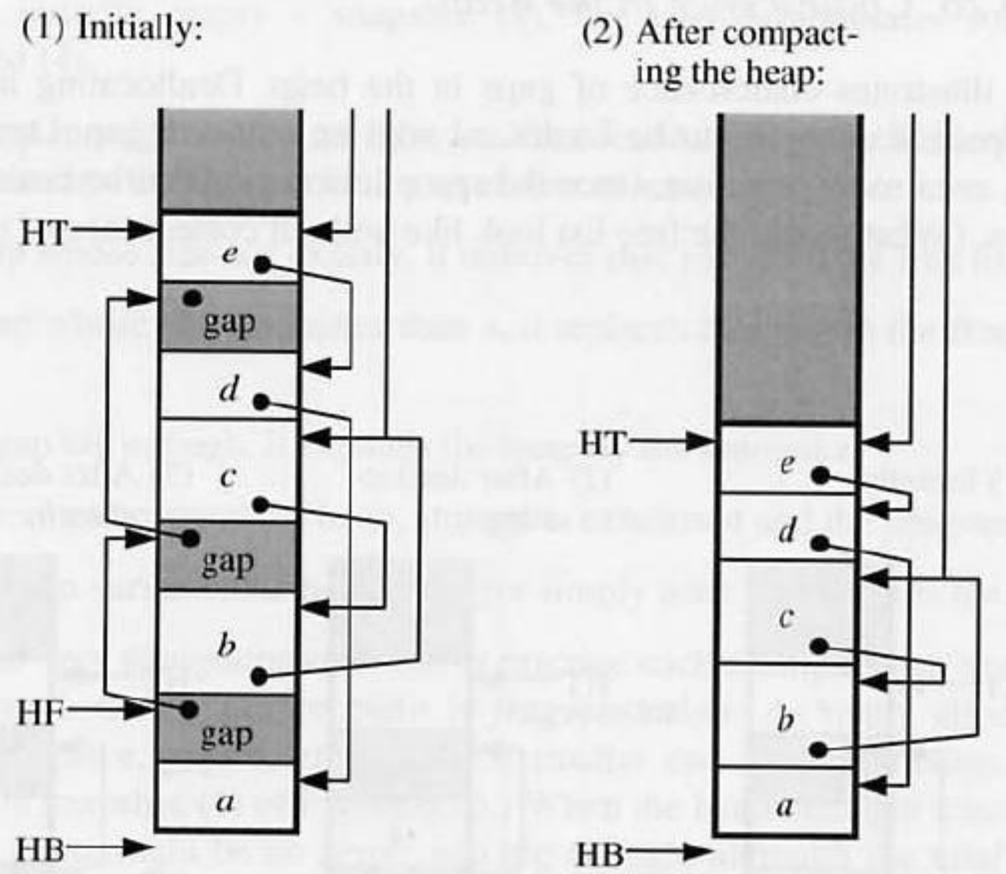
1. Manter uma lista de blocos livres

    1. Alocar no primeiro bloco livre com tamanho que seja suficiente para a nova variável.
    2. Alocar no primeiro bloco livre com o menor tamanho mas que seja suficiente para a nova variável.

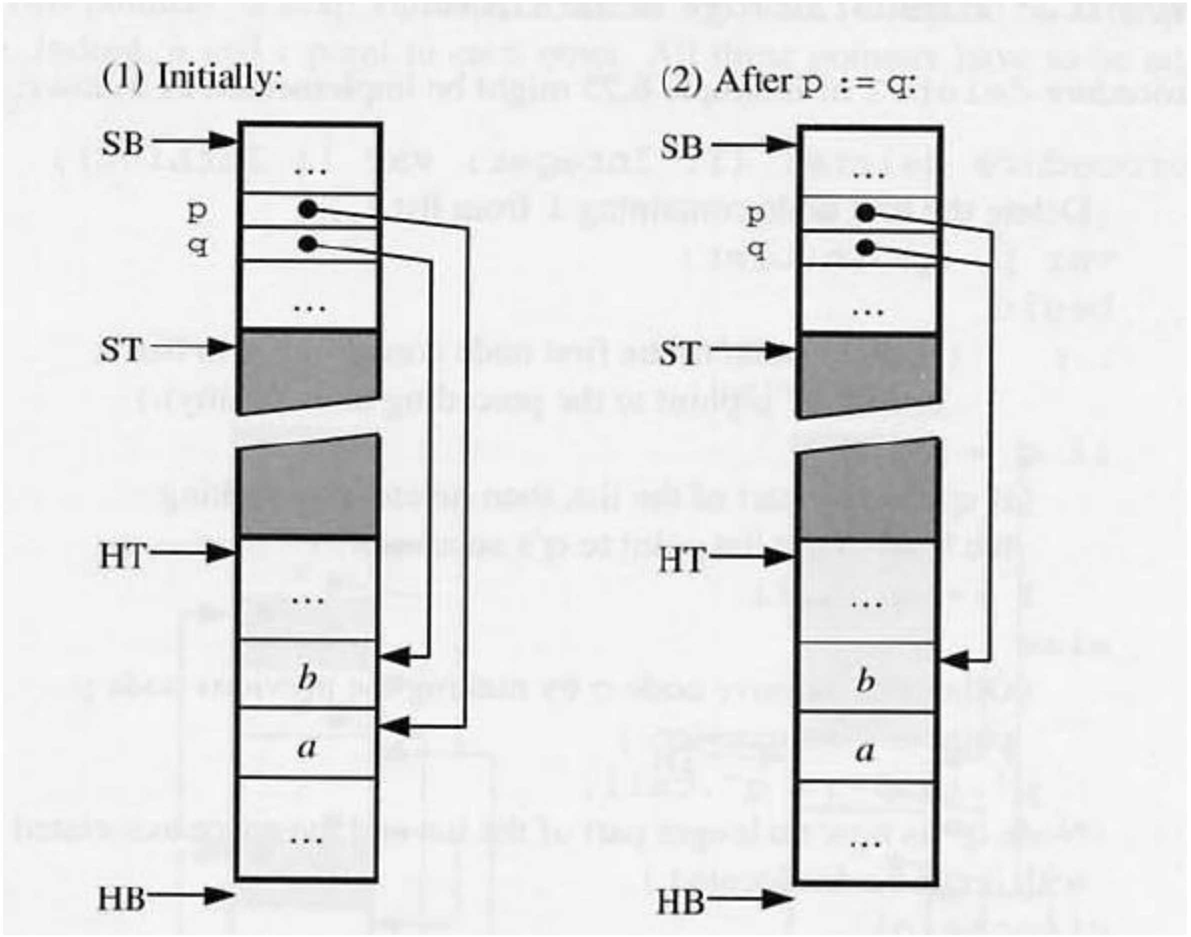
2. Fazer coalescência de blocos livres.

3. Compactar o heap.


(1) Initially:
(2) After dealloc-ating c:
(3) After dealloc-ating b:
HT
e
gap
d
c
gap
b
HF
gap
a
HB
HT
e
gap
d
gap
b
HF
gap
a
HB
HT
e
gap
d
HF
gap
a
HB

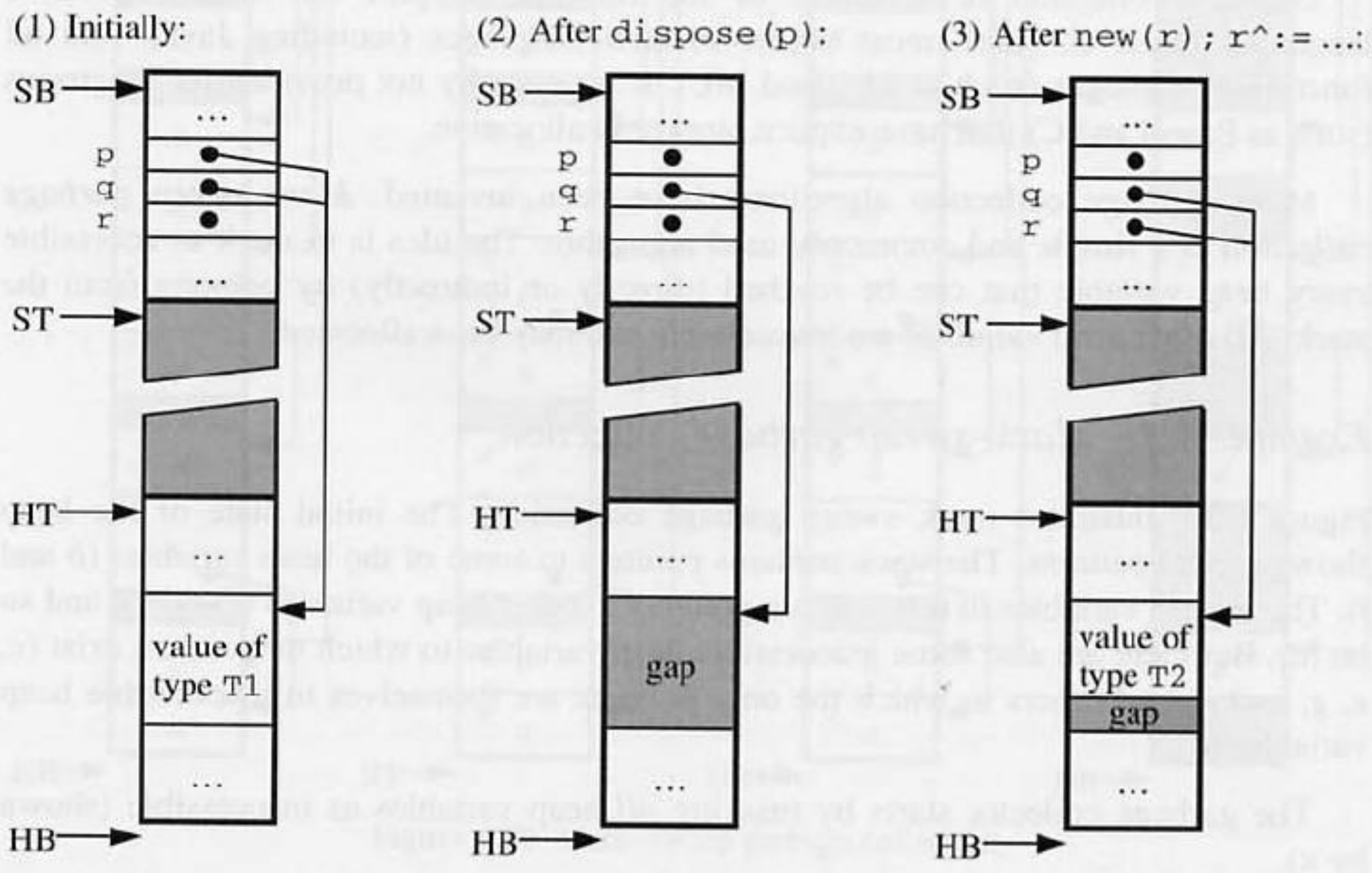

(1) Initially:
(2) After compacting the heap:
HT
e
gap
d
c
gap
b
HF
gap
a
HB
HT
e
d
c
b
a
HB

```
procedure deleteI (i: Integer; var l: IntList);
    {Delete the first node containing i from list l.}
    var p, q: IntList;
    begin
    ...;     {Make q point to the first node containing i in list l,
              and make p point to the preceding node (if any).}
    if q = l then
        {If q is at the start of the list, then delete it by making
         the head of the list point to q's successor. }
        l := q^.tail
    else
        {Otherwise remove node q by making the previous node p
         point to q's successor. }
        p^.tail := q^.tail;
    {Node q^ is now no longer part of the list and the space associated
     with it can be deallocated.}
    dispose(q)
    end {deleteI}
```

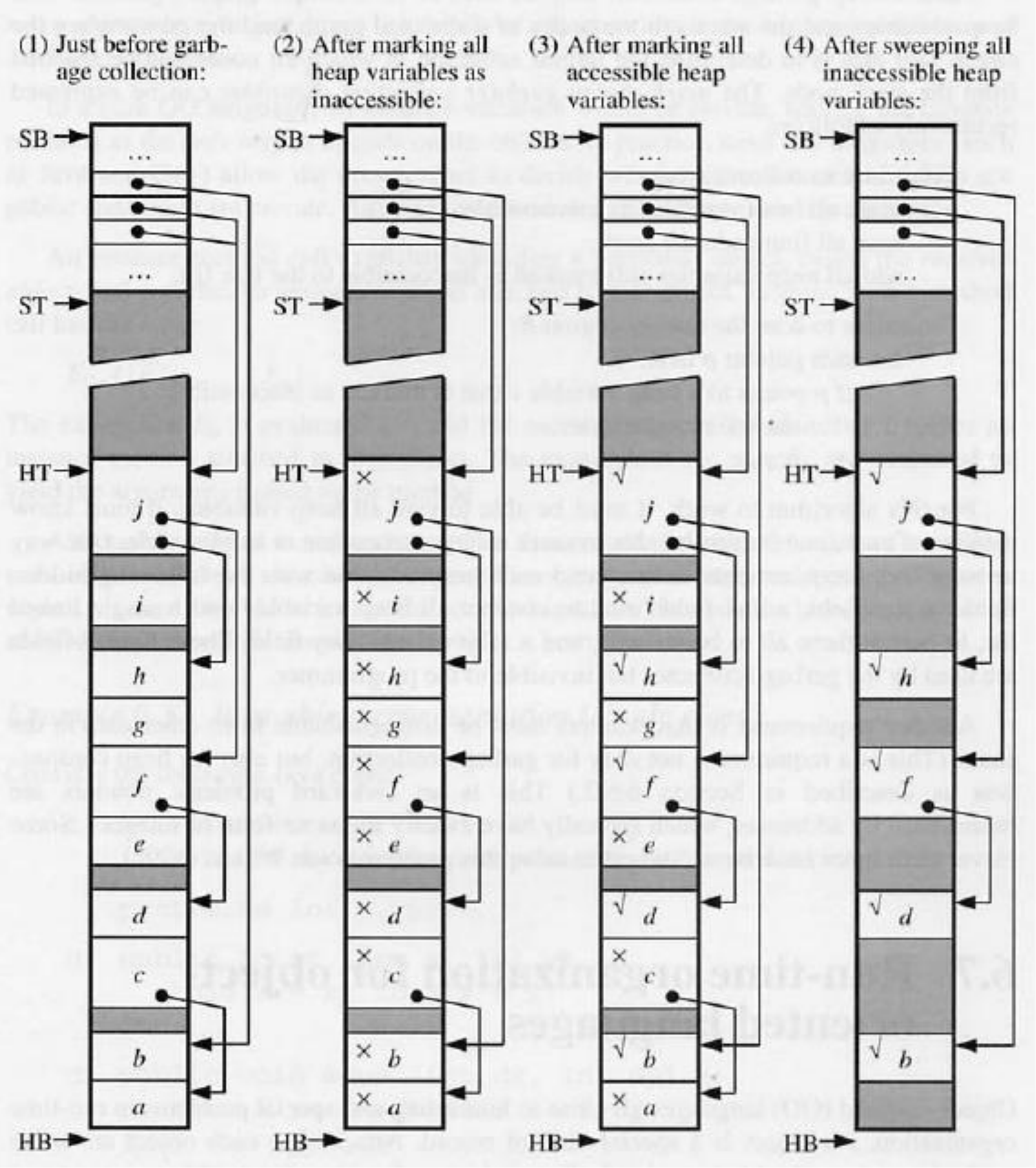


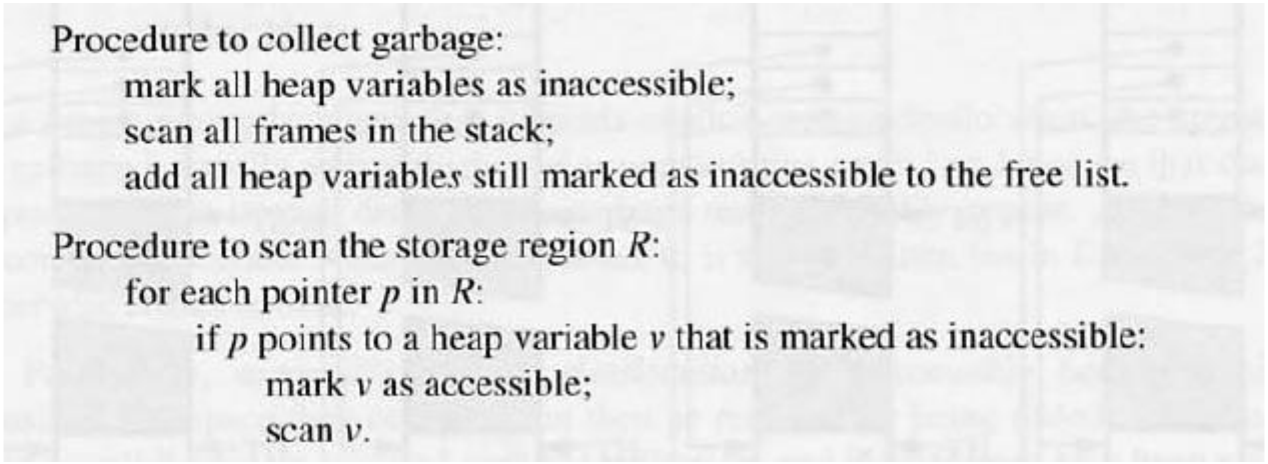
(1) Initially:
SB
...
p
q
...
ST
HT
...
b
a
...
HB
(2) After p := q:
SB
...
p
q
...
ST
HT
...
b
a
...
HB

```
var p, q: ^T1; r: ^T2;
...
new(p);   p^ := value of type T1;
q := p;
...;
dispose(p);                                   (2)
...;
new(r);   r^ := value of type T2;             (3)
...;
q^ := value of type T1;                       (4)
```



(1) Initially:
(2) After dispose(p):
(3) After new(r); r^:= ...:
SB
ST
HT
HB
p
q
r
...
value of type T1
gap
value of type T2


(1) Just before garbage collection:
(2) After marking all heap variables as inaccessible:
(3) After marking all accessible heap variables:
(4) After sweeping all inaccessible heap variables:
SB
ST
HT
HB
j
i
h
g
f
e
d
c
b
a



```
Procedure to collect garbage:
    mark all heap variables as inaccessible;
    scan all frames in the stack;
    add all heap variables still marked as inaccessible to the free list.

Procedure to scan the storage region $R$:
    for each pointer $p$ in $R$:
        if $p$ points to a heap variable $v$ that is marked as inaccessible:
            mark $v$ as accessible;
            scan $v$.
```