Melhores momentos

AULA 5

Pausa para nossos comerciais

- Plantões de dúvidas: Vitor

  Horário: segundas das 13h às 14h

  Onde: nas mesas do térreo do CCSL

- Plantões de dúvidas: Matheus

  Horário: quartas das 13h às 14h

  Onde: Sala 9 do Bloco B

Conceitos

Endereços: a memória é um vetor e o índice desse vetor onde está uma variável é o endereço da variável.

Com o operador `&` obtemos o endereço de uma variável.

Exemplos:

- `&i` é o endereço de `i`
- `&ponto` é o endereço da estrutura `ponto`
- `&v[2]` é o endereço de `v[2]`

Conceitos

Ponteiros: são variáveis que armazenam endereços.

Exemplos:

```
int *p;  /* ponteiro para int*/
char *q;  /* ponteiro para char*/
double *r;  /* ponteiro para double*/
```

Conceitos

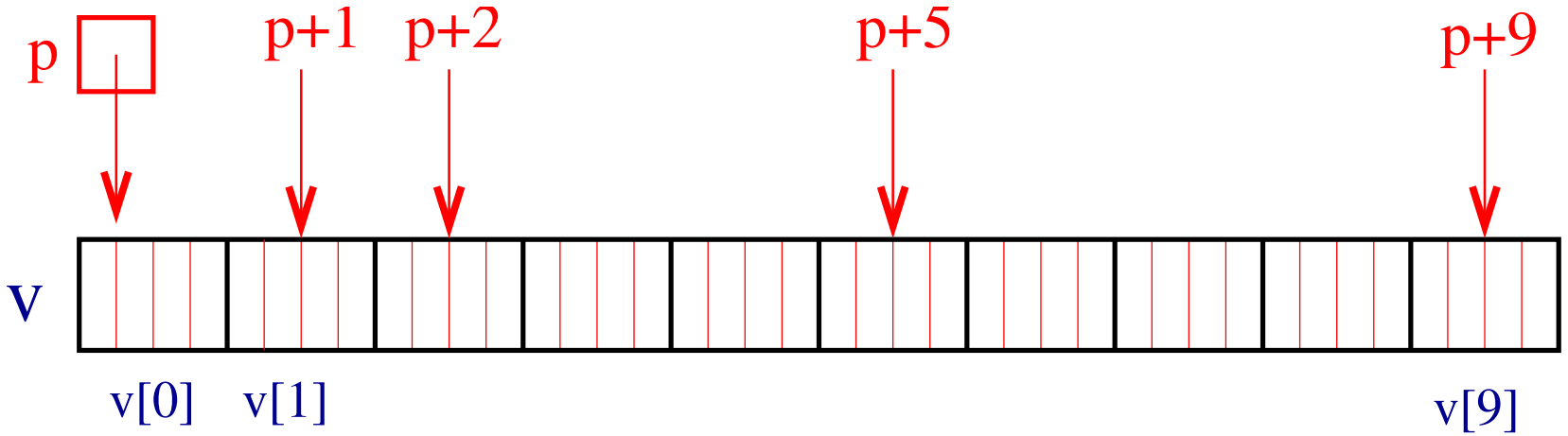
Dereferenciação: Se `p` aponta para a variável `i`, então `*p` é sinônimo de `i`.

Exemplo:

```
p = &i;   /* p aponta para i/
(*p)++;   é o mesmo que   i++;
```

Conceitos

Aritmética de ponteiros: se `p` é um apontador para um `int` e o seu conteúdo é 64542, então `p+1` é 64546, pois um `int` ocupa 4 bytes (no meu computador...).



Conceitos

Vetores e ponteiros: o nome de um vetor é sinônimo do endereço da posição inicial do vetor.

Exemplo:

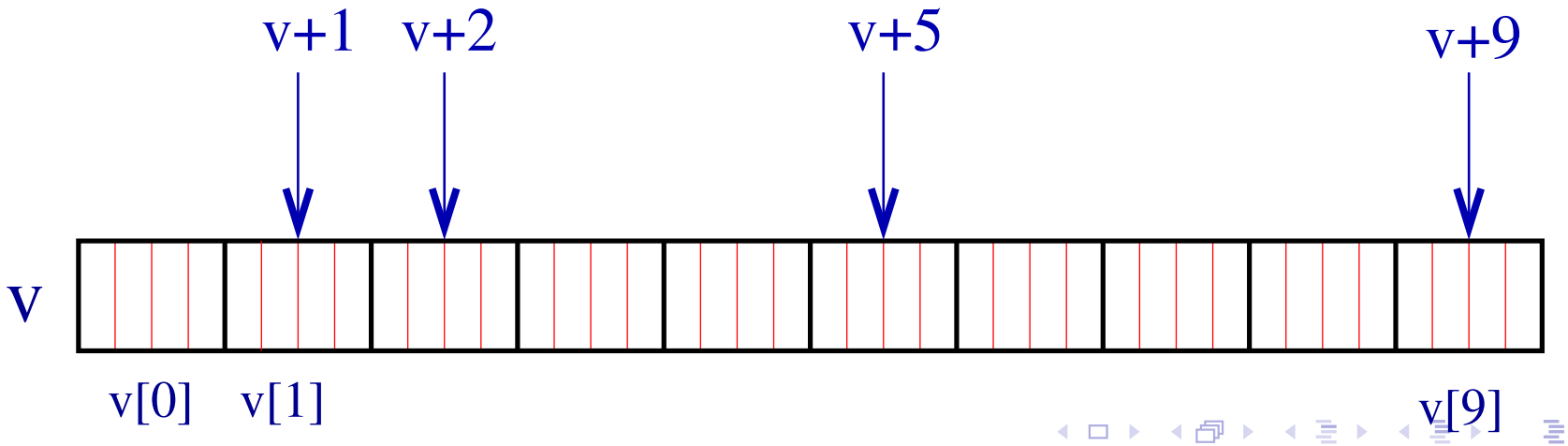
```
int v[10];
```

`v` é sinônimo de `&v[0]`

`v+1` é sinônimo de `&v[1]`

`v+2` é sinônimo de `&v[2]`

. . .

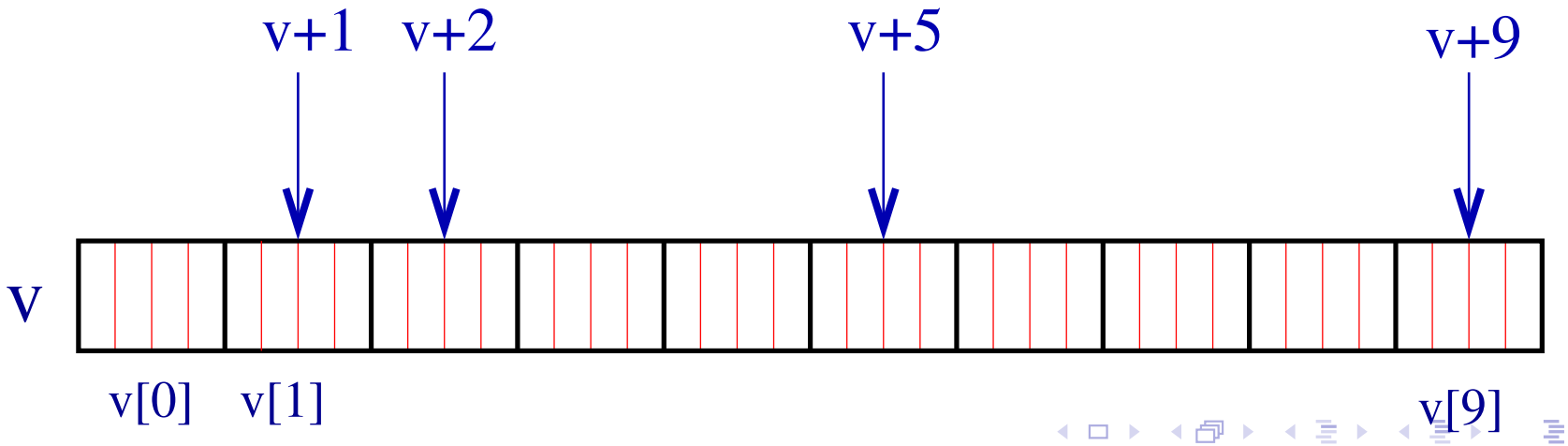


Conceitos

Vetores e ponteiros: o nome de um vetor é sinônimo do endereço da posição inicial do vetor.

Exemplo:

```
int v[10];
*v é sinônimo de v[0]
*(v+1) é sinônimo de v[1]
*(v+2) é sinônimo de v[2]
. . .
```



Vetores como parâmetros

Como parâmetros formais de uma função,

```
char s[ ];
```

e

```
char *s;
```

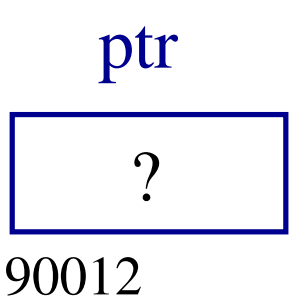
são equivalentes. O Kernighan e Ritchie preferem a segunda pois diz mais explicitamente que a variável é um apontador.
Outro exemplo

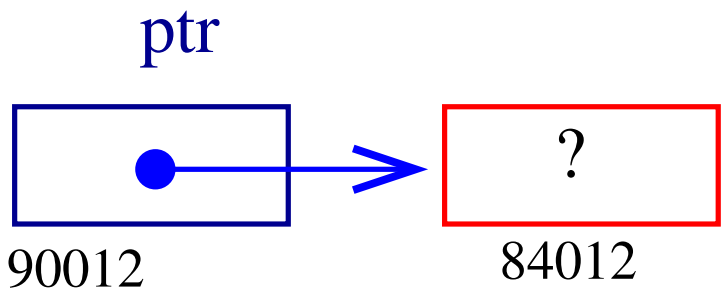
```
int main(int argc, char **argv);
```

malloc

A função `malloc` aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco.

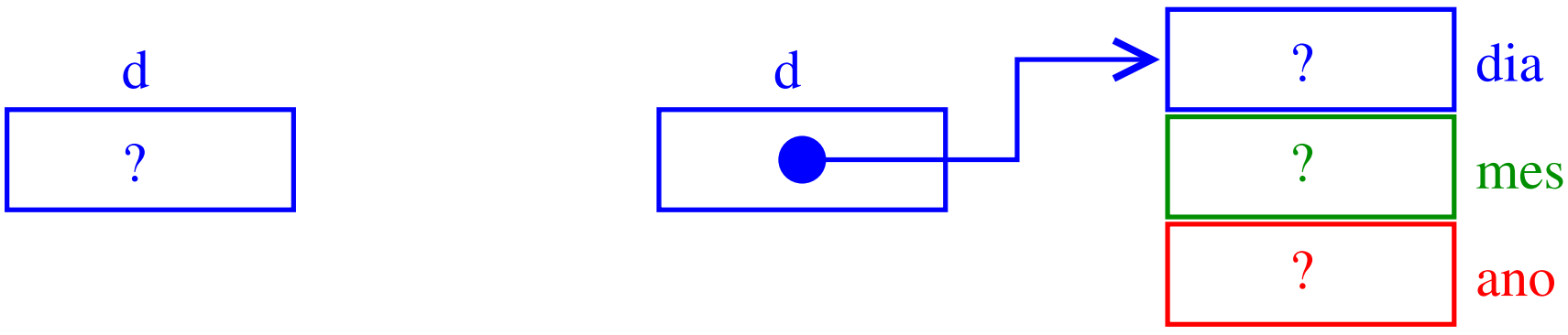
```
char *ptr;
ptr = malloc(1);
scanf("%c",ptr);
```





malloc

```
typedef struct {
    int dia,mes,ano;
} Data;
Data *d;
d = malloc(sizeof(Data));
```



malloc

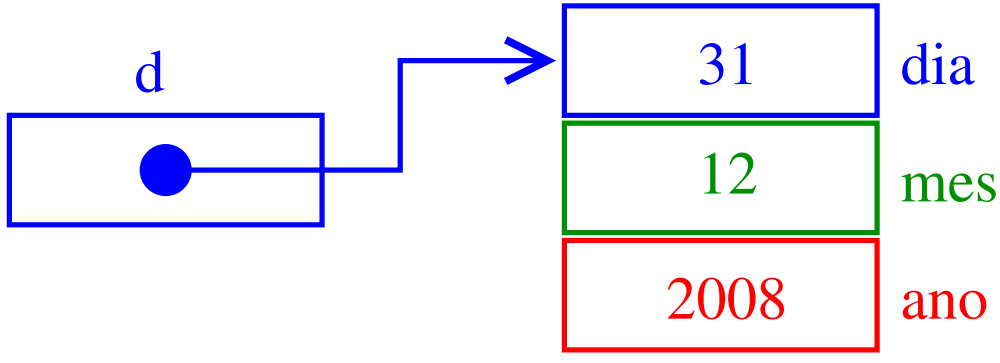
Se `p` é ponteiro para uma estrutura então

```
p->campo-da-estrutura
```

é uma abreviatura de

```
(*p).campo-da-estrutura
```

```
d->dia=31; d->mes=12; d->ano=2008;
```



AULA 6

Hoje

- alocação dinâmica de memória
- listas em vetores
- listas encadeadas em vetores

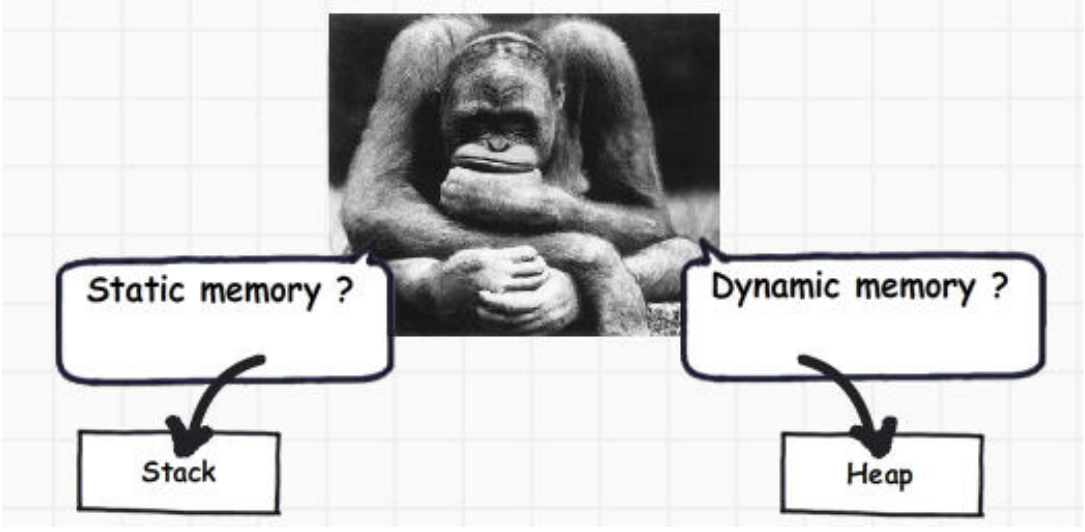
Alocação dinâmica de memória

PF Apêndice F

http://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html

The C programming Language
Brian W. Kernighan e Dennis M. Ritchie
Prentice-Hall

Alocação dinâmica



Fonte: http://www.codeproject.com/

Alocação dinâmica

Às vezes, a quantidade de memória que o programa necessita só se torna conhecida durante a execução do programa.

Para lidar com essa situação é preciso recorrer à **alocação dinâmica de memória**.

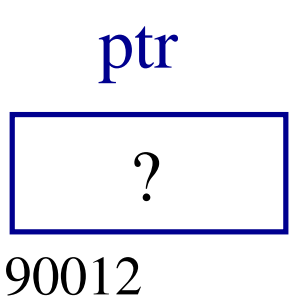
A alocação dinâmica é gerenciada pelas funções `malloc` e `free`, que estão na biblioteca `stdlib`

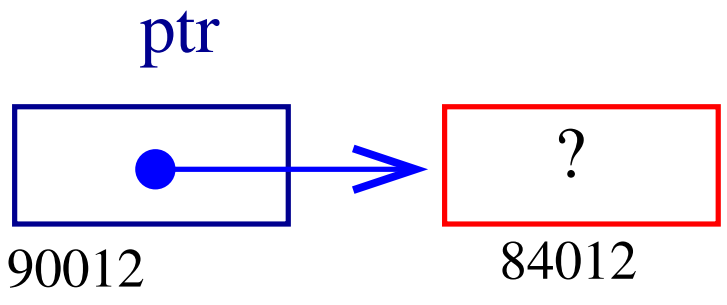
```
#include <stdlib.h>
```

malloc

A função `malloc` aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco.

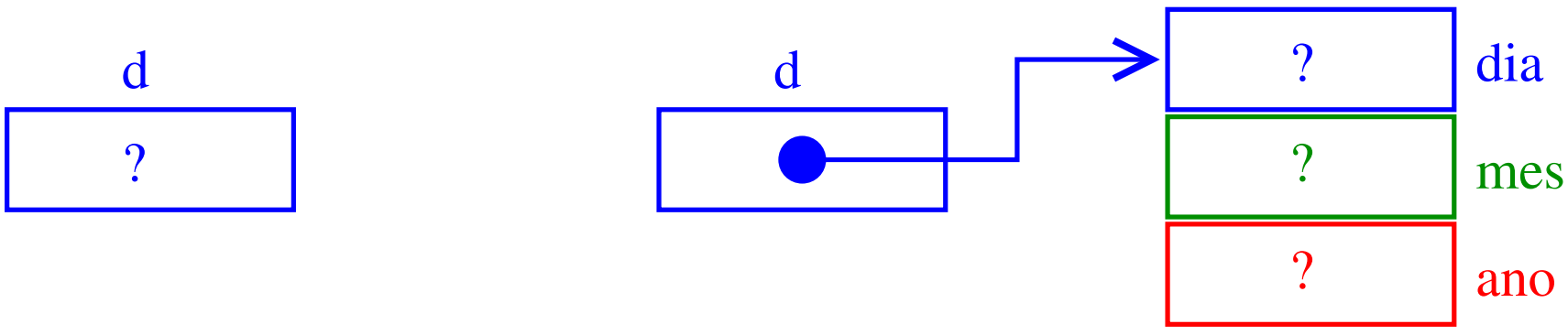
```
char *ptr;
ptr = malloc(1);
scanf("%c",ptr);
```





malloc

```
typedef struct {
    int dia,mes,ano;
} Data;
Data *d;
d = malloc(sizeof(Data));
```



malloc

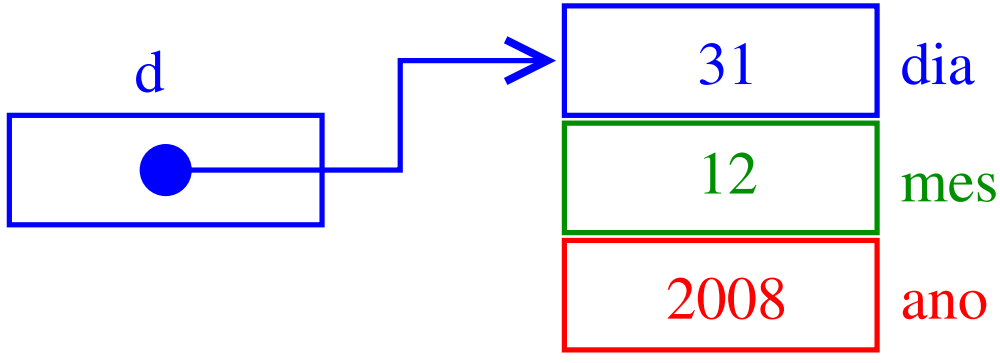
Se `p` é ponteiro para uma estrutura então

```
p->campo-da-estrutura
```

é uma abreviatura de

```
(*p).campo-da-estrutura
```

```
d->dia=31; d->mes=12; d->ano=2008;
```



A memória é finita

Se malloc não consegue alocar mais espaço então retorna NULL.

```
ptr = malloc(sizeof(Data));
if (ptr == NULL) {
   printf("Socorro!  malloc devolveu NULL!\n");
   exit(EXIT_FAILURE);
}
```

A memória é finita

É conveniente usarmos a função

```
void *mallocSafe (int nbytes) {
   void *ptr;
   ptr = malloc(nbytes);
   if (ptr == NULL) {
        printf("Socorro!  malloc devolveu "
                "NULL!\n");
        exit(EXIT_FAILURE);
   }
   return ptr;
}
```

free

Fonte: http://www.zazzle.com.br/

A função `free` libera a memória alocada por `malloc`.

```
free(d);
```

Há pessoas que, por questões de segurança, gostam de atribuir `NULL` a um ponteiro depois da liberação de memória.

```
free(d);
d = NULL;
```

Vetores dinâmicos

```
int *v; int i, n;

printf("Digite o tamanho do vetor:  ");
scanf("%d", &n);

v = mallocSafe(n*sizeof(int));

for (i = 0; i < n; i++)
       *(v+i) = i; /* v[i] = i; */

for (i = 0; i < n; i++)
   printf("end.  v[%d] = %p cont v[%d] = %d\n",
             i, (void*)(v+i), i, v[i]);
free(v);
```

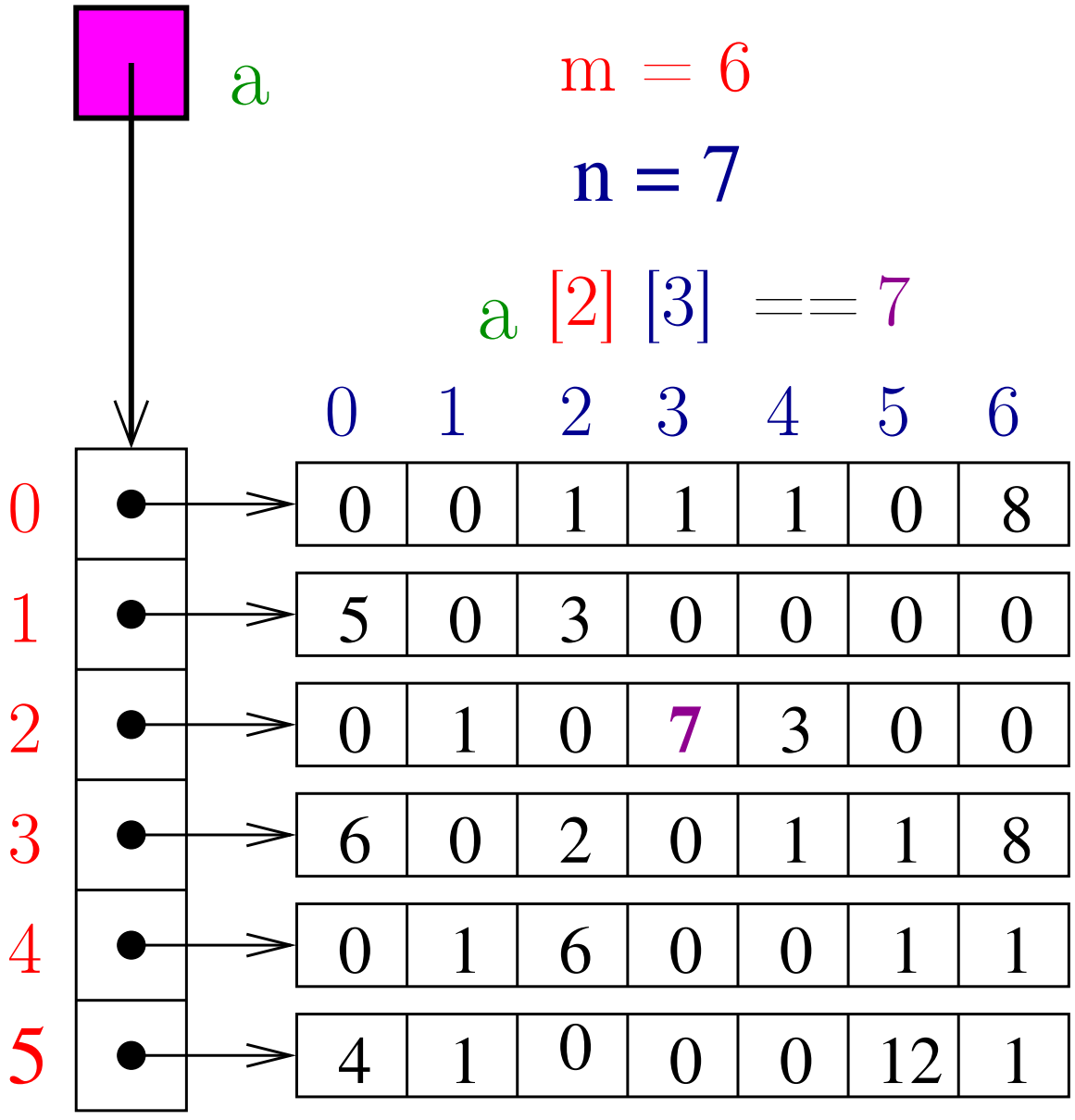
Matrizes dinâmicas

Matrizes bidimensionais são implementadas como vetores de vetores.

```
int **a;
int i;
a = mallocSafe(m * sizeof(int*));
for (i = 0; i < m; ++i)
   a[i] = mallocSafe(n * sizeof(int));
```

O elemento de `a` que está na linha `i` e coluna `j` é `a[i][j]`.

Matrizes dinâmicas



Liberação de memória de matrizes

Para liberarmos a memória alocada dinamicamente para uma matriz devemos seguir os passos inversos aos da alocação trocando `mallocSafe` por `free`.

Liberação de memória de matrizes

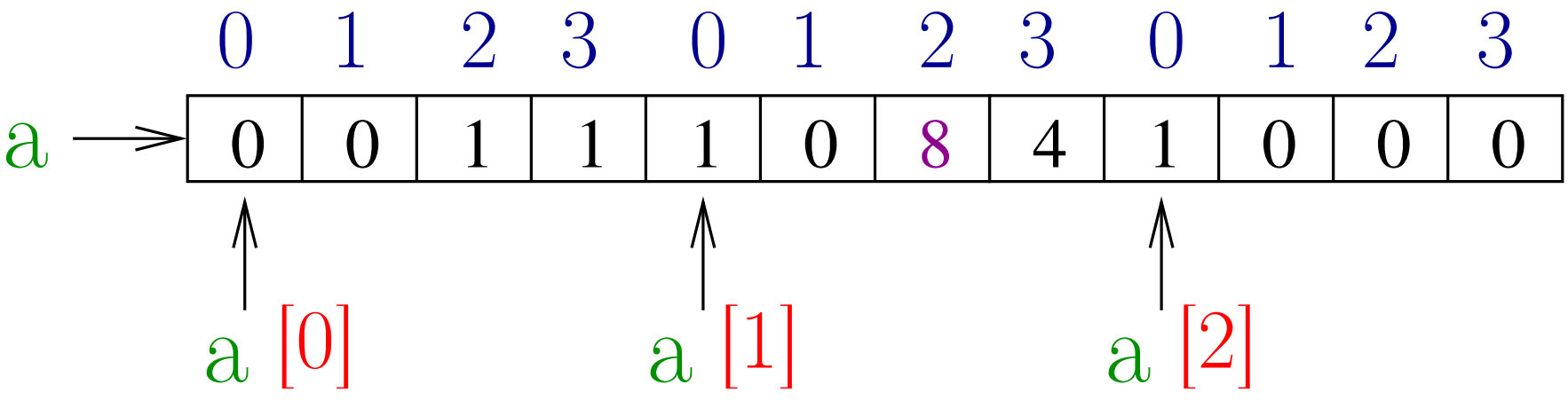
Para liberarmos a memória alocada dinamicamente para uma matriz devemos seguir os passos inversos aos da alocação trocando `mallocSafe` por `free`.

```
void freeMatrizInt(int **a) {
   int i;

   for (i = 0; i < m; i++){
       free(a[i]); /* libera a linha i */

       a[i] = NULL;
   }
   free(a);/* libera vetor de ponteiros */
   a = NULL;
}
```

Matrizes automáticas

```
int a[3][4];
```

a [1] [2] ==8



Passagem de parâmetros

Suponha que temos os protótipos de funções

```
void f(int **m);
int g(int m[][64]);
```

e as declarações

```
int **a;
int m[16][64];
```

Passagem de parâmetros

Suponha que temos os protótipos de funções

```
void f(int **m);
int g(int m[][64]);
```

e as declarações

```
int **a;
int m[16][64];
```

então temos que

```
f(a);          /* ok */
i = g(a);      /* erro */
i = g(m);      /* ok */
f(m);          /* erro */
```

Listas em vetores

PF 3

http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html

Lista de nomes em ordem alfabética

$n = 8$

Remover Joao

$n = 8$

Remover Joao

Remover Joao

Remover Joao

Remover Joao

Remover Joao

Busca em um vetor

A função recebe `x`, `n` ≥ 0 e `v` e devolve um índice `k` em $0\,..\,$`n`-1 tal que `x == v[k]` .
Se tal `k` não existe, devolve -1

```
int busca(int x, int n, int v[])
{
   int k;
   k = n-1;
   while (k >= 0 && v[k] != x)
       k -= 1;
   return k;
}
```

Busca recursiva em vetor

A função recebe `x`, `n` ≥ 0 e `v` e devolve um índice `k` em $0\,..\,$`n`-1 tal que `x == v[k]` .
Se tal `k` não existe, devolve -1

```
int buscaR(int x, int n, int v[])
{
   if (n == 0) return -1;
   if (x == v[n-1]) return n-1;
   return buscaR(x, n-1, v);
}
```

Conclusões

No pior caso o consumo de tempo da função `busca` é proporcional a `n`.

O consumo de tempo da função `busca` é $O(n)$.

$O(n)$ = "é da ordem de `n`"

Remoção em vetor

Esta função recebe $0 \leq$ `k` $<$ `n` e remove o elemento `v[k]` do vetor `v[0 . . n-1]` .
A função devolve o novo valor de `n`.

```
int remove(int k, int n, int v[])
{
   int j;
   for (j = k+1; j < n; j++)
       v[j-1] = v[j];
   return n-1;
}
```

Remoção recursiva

A função recebe $0 \leq$ `k` $<$ `n` e remove o elemento `v[k]` do vetor `v[0 . . n-1]`.
A função devolve o novo valor de `n`.

```
int removeR(int k, int n, int v[])
{
   if (k == n-1) return n-1;
   v[k] = v[k+1];
   return removeR(k+1, n, v);
}
```

Conclusões

No pior caso o consumo de tempo da função `remove` é proporcional a `n`.

O consumo de tempo da função `remove` é $O(n)$.

$O(n)$ = "é da ordem de `n`"

Inserir Walter

Inserir Walter

$n = 8$

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Luiz
4
Maria
5
Rui
6
Sergio
7
Walter
8
9
10
n = 8

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Luiz
4
Maria
5
Rui
6
Sergio
7
Walter
8
9
10
n = 9

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Luiz
4
Maria
5
Rui
6
Sergio
7
8
Walter
9
10
n = 9

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Luiz
4
Maria
5
Rui
6
7
Sergio
8
Walter
9
10
n = 9

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Luiz
4
Maria
5
6
Rui
7
Sergio
8
Walter
9
10
n = 9

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Luiz
4
5
Maria
6
Rui
7
Sergio
8
Walter
9
10
n = 9

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Walter
9
10
n = 9

Inserir Ana


0
Carlos
1
Eduardo
2
3
Helio
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Walter
9
10
n = 9

Inserir Ana


0
Carlos
1
2
Eduardo
3
Helio
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Walter
9
10
n = 9

Inserir Ana

$n = 9$

Inserir Ana

$n = 9$

Inserção em um vetor

Esta função insere `x` entre `v[k-1]` e `v[k]` no vetor `v[0 . . n-1]`. Ela supõe apenas que $0 \leq k \leq n$. A função devolve o novo valor de `n`.

```
int insere(int k, int x, int n, int v[])
{
   int j;
   for (j = n; j > k; j--)
       v[j] = v[j-1];
   v[k] = x;
   return n+1;
}
```

Inserção recursiva

Recebe $0 \leq$ `k` $\leq$ `n` e insere `x` entre `v[k-1]` e `v[k]` no vetor `v[0 . . n-1]`. A função devolve o novo valor de `n`. (Hmmm. Essa função devia se `void`...)

```
int insereR(int k, int x, int n, int v[])
{
   if (k == n) v[n] = x;
   else {
       v[n] = v[n-1];
       insereR(k, x, n-1, v);
   }
   return n+1;
}
```

Conclusões

No pior caso o consumo de tempo da função `insere` é proporcional a `n`.

O consumo de tempo da função `insere` é $O(n)$.

$O(n)$ = "é da ordem de `n`"

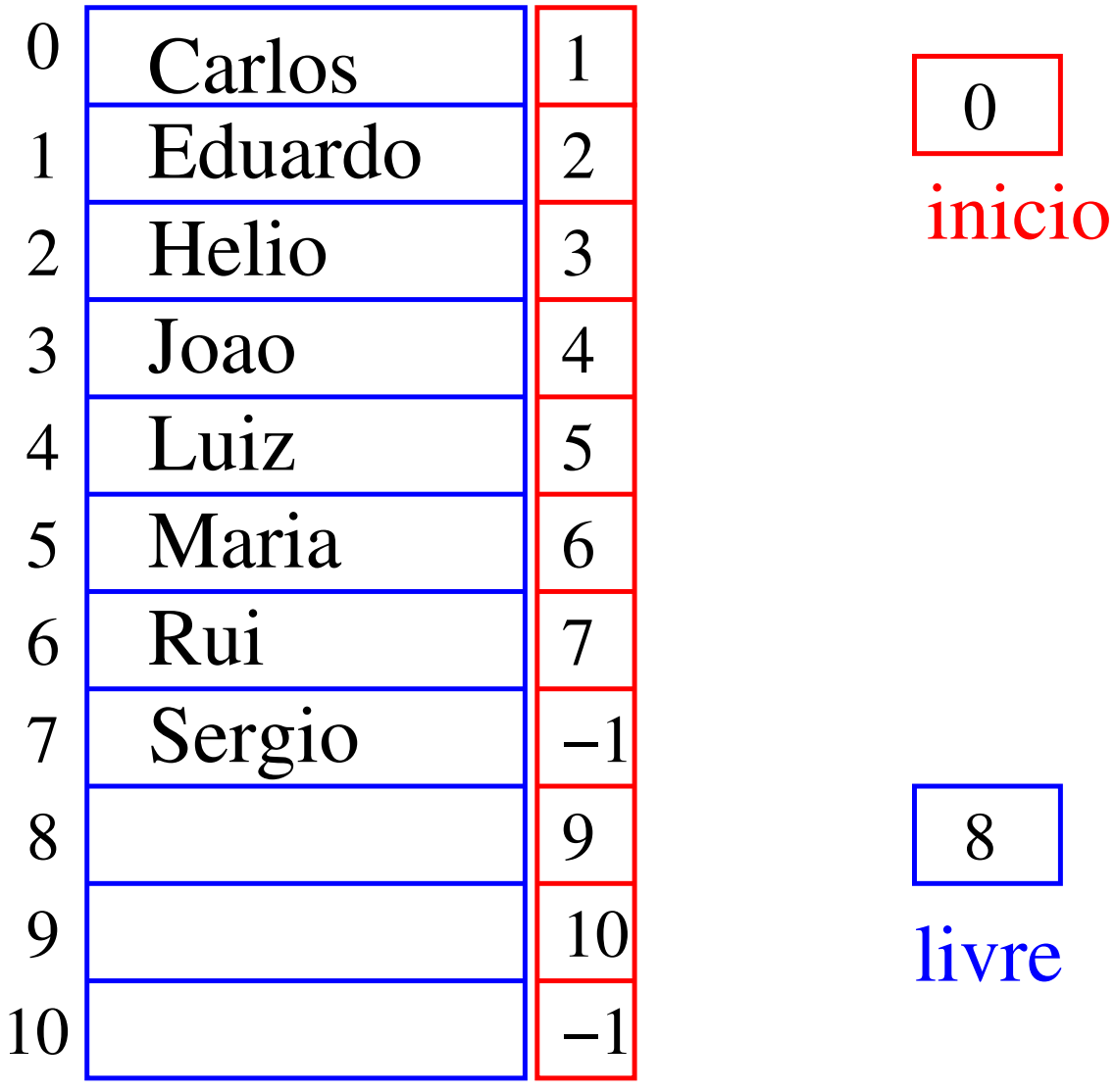
Mais conclusões

Manter uma lista em um vetor sujeita a remoções e inserções pode dar muito trabalho com movimentações.

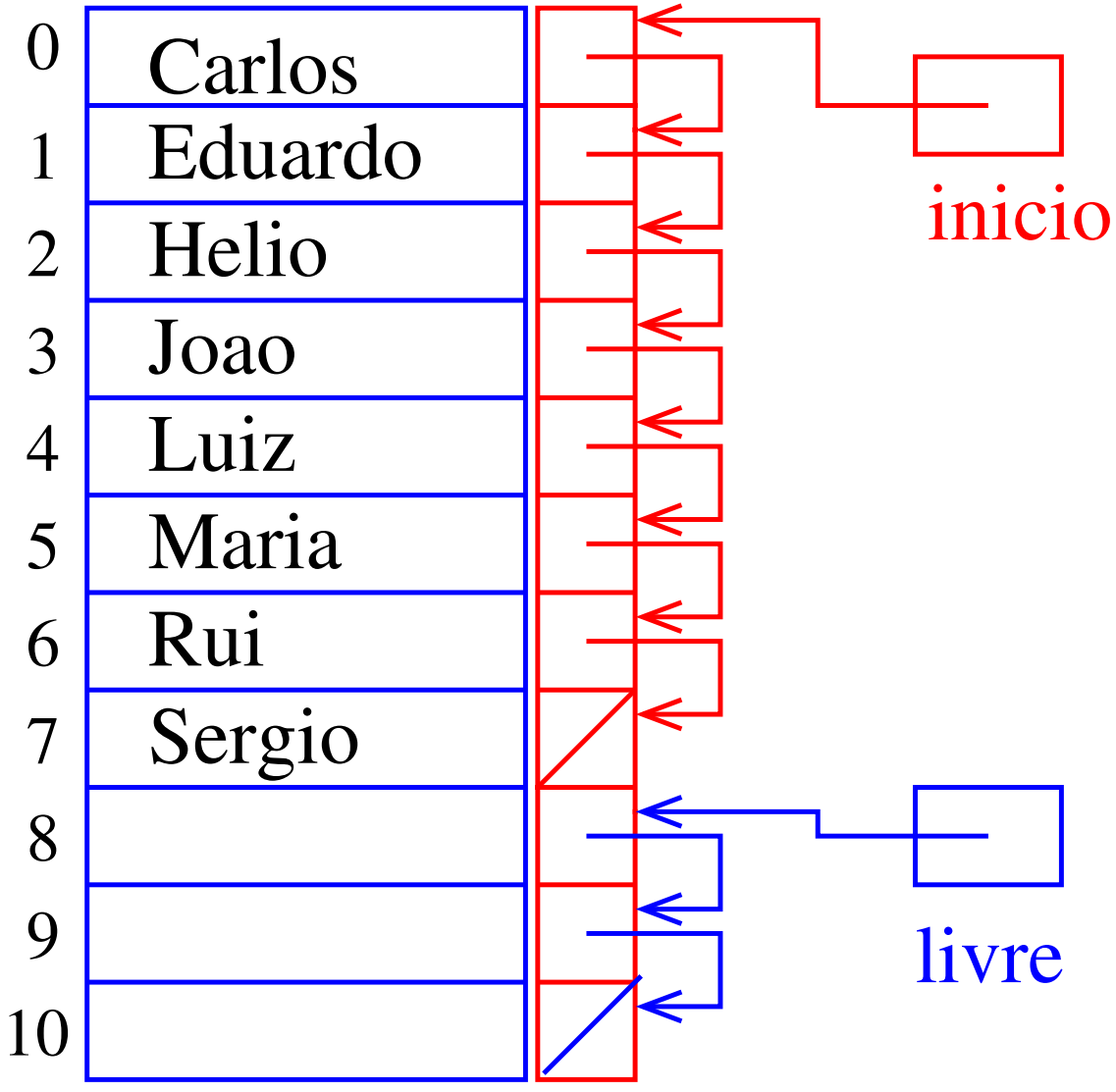
Veremos uma maneira alternativa que pode dar menos trabalho com movimentações, se estivermos dispostos a gastar um pouco mais de espaço.

Listas encadeadas em vetores

Lista encadeada



Lista encadeada



Estrutura de uma lista encadeada em vetor

Uma **lista encadeada** (= *linked list* = lista ligada) é uma sequência de células; cada célula contém um objeto de algum tipo e o endereço da célula seguinte.

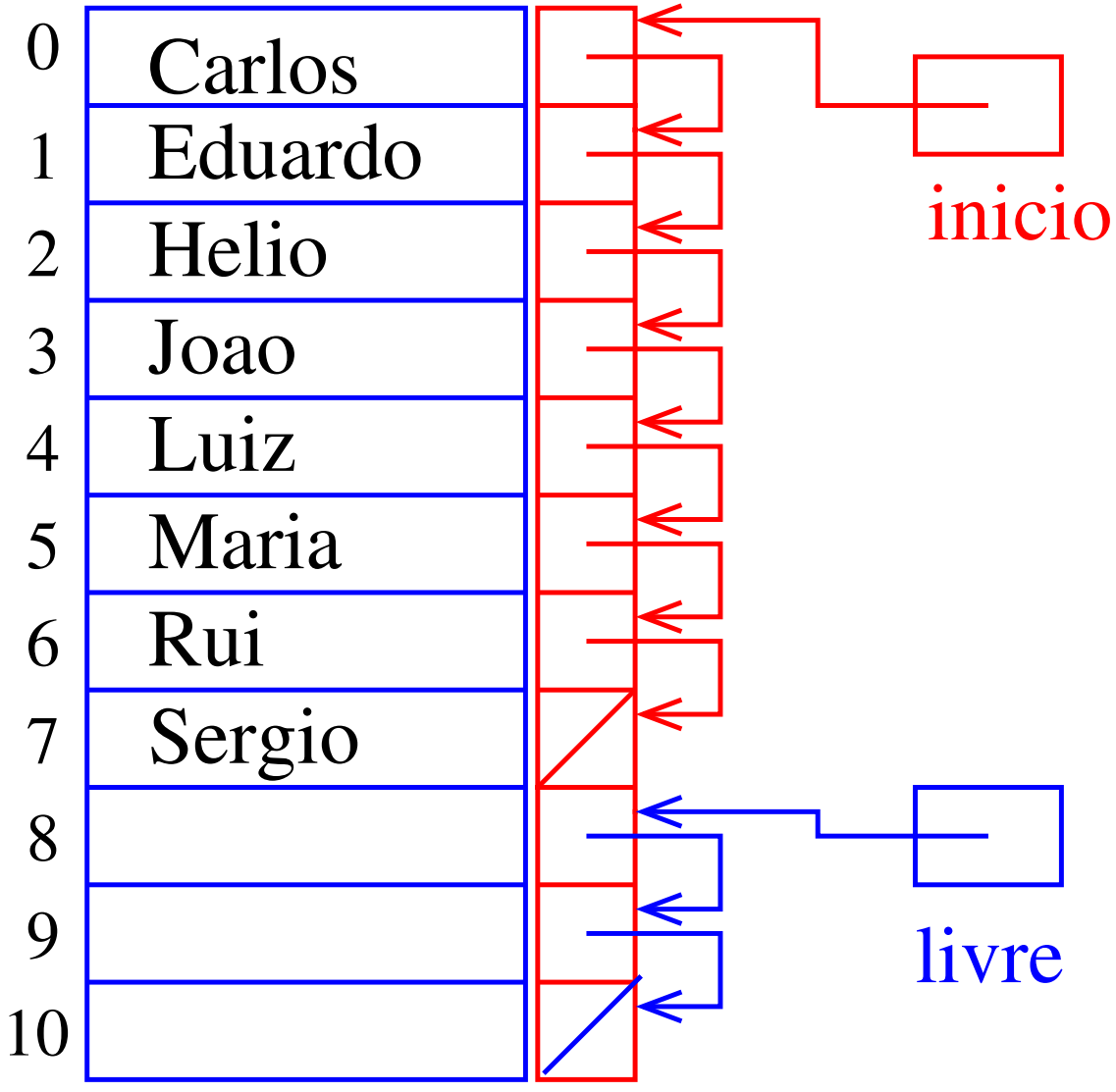
```
struct celula {
   int conteudo;
   int prox;
 };
typedef struct celula Celula;
Celula v[MAX];
int inicio;
int livre;
```

Imprime conteúdo de uma lista

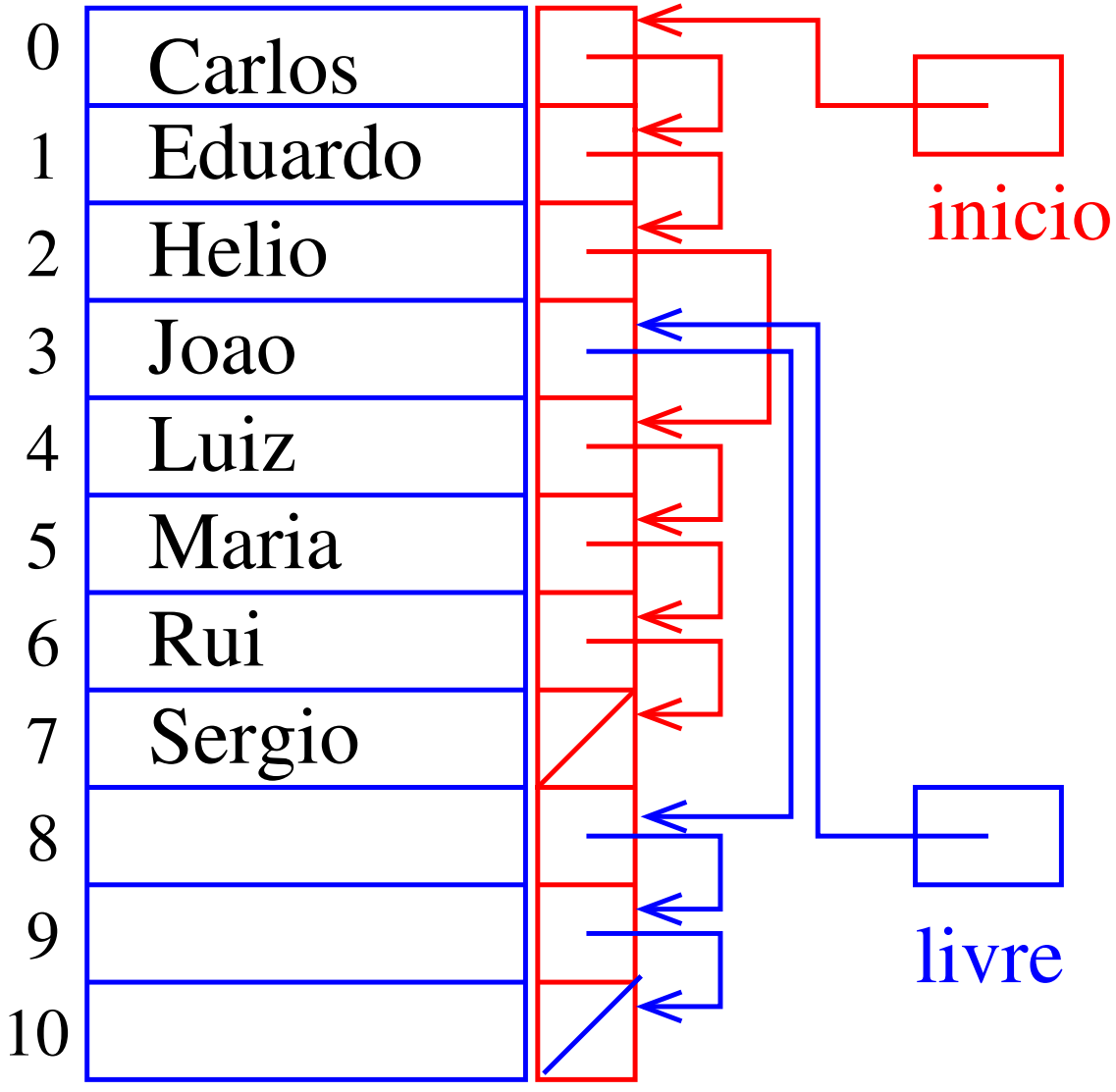
Esta função recebe o índice inicio de uma lista encadeada em um vetor e imprime os elementos da lista.

```
# define NULO -1
void imprime(int inicio, Celula v[])
{
   int p;
   for (p =inicio; p != NULO; p=v[p].prox)
       printf("%d ", v[p].conteudo);
   printf("\n");
}
```

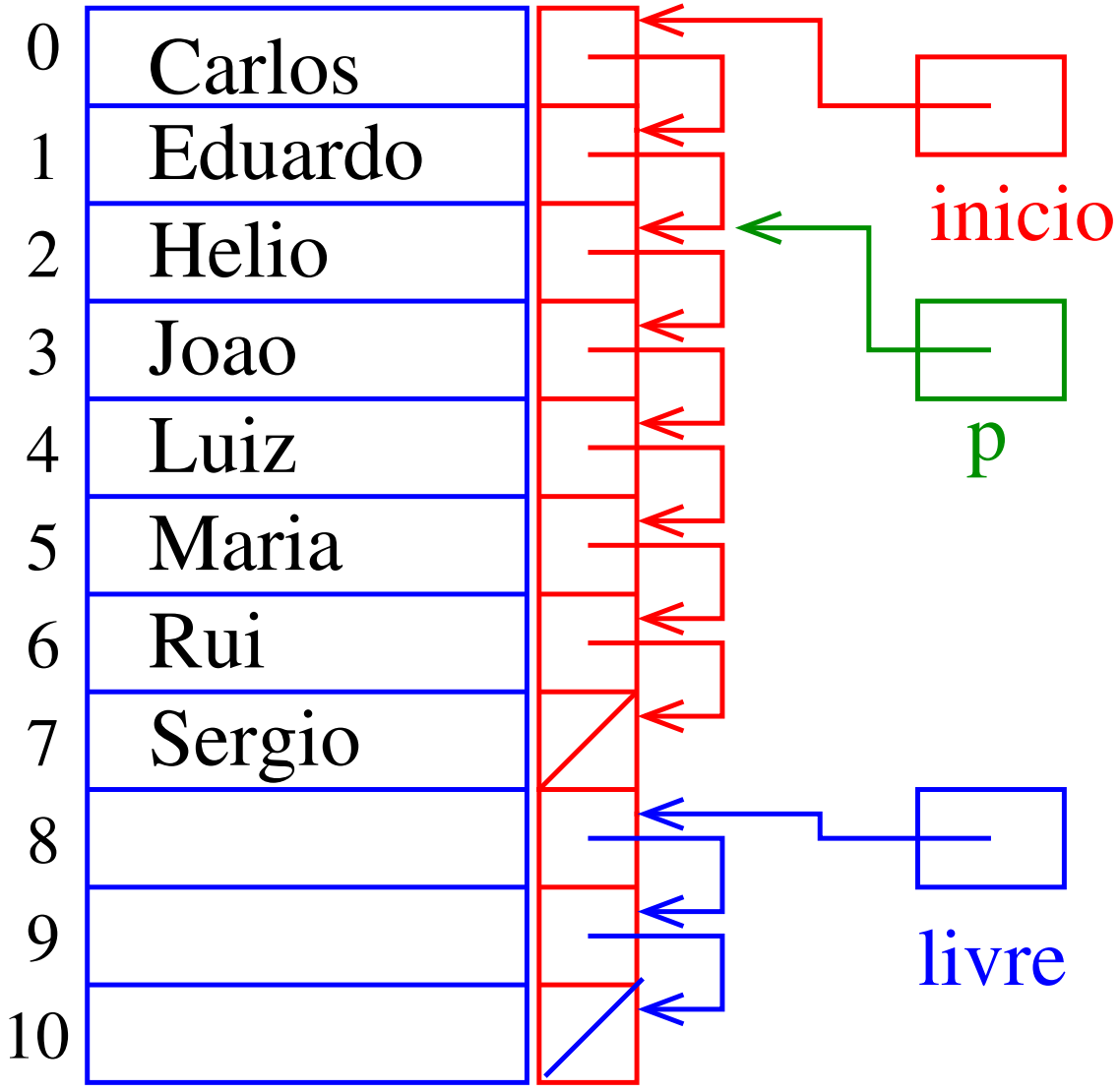
Remover Joao



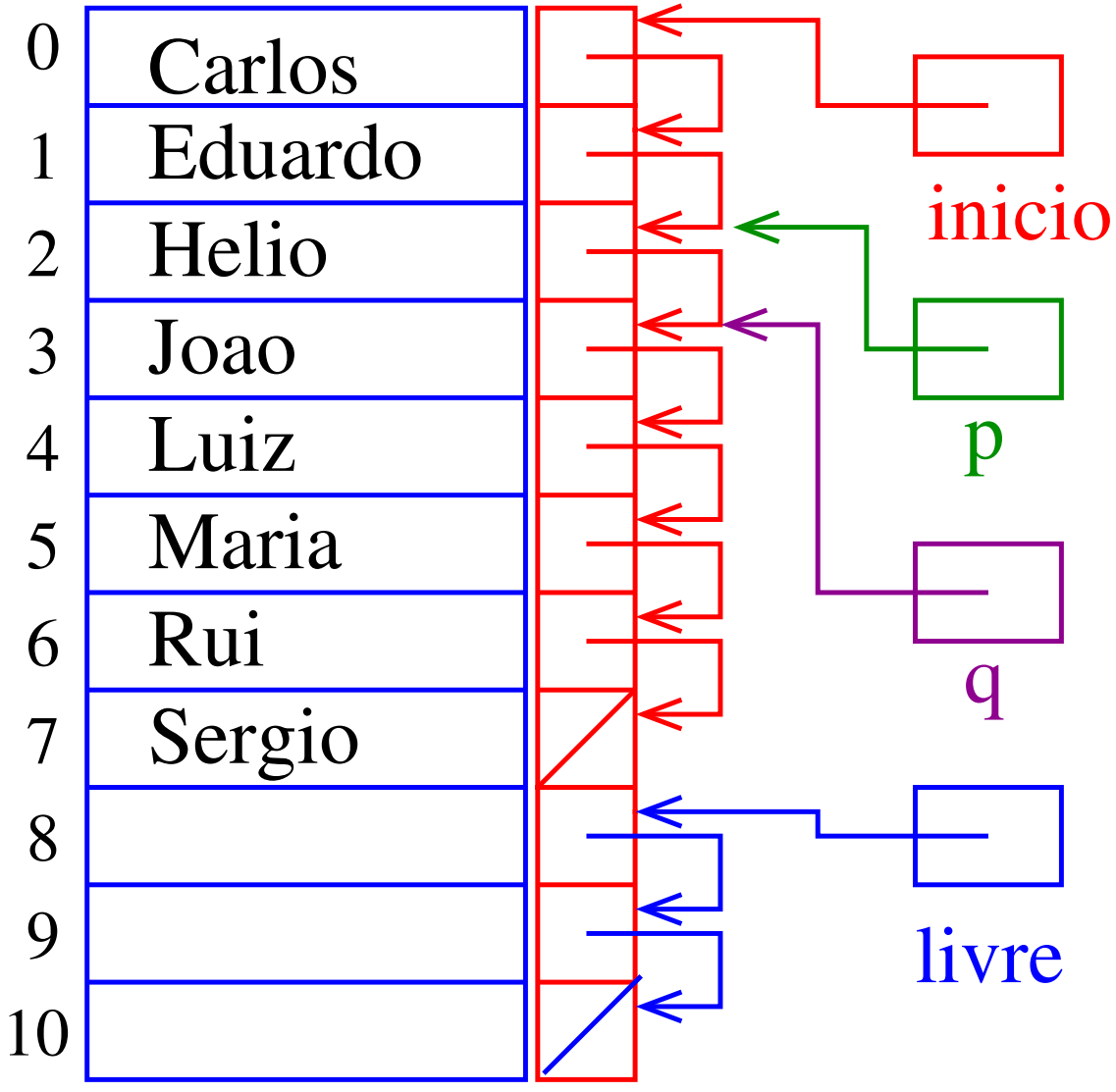
Remover Joao



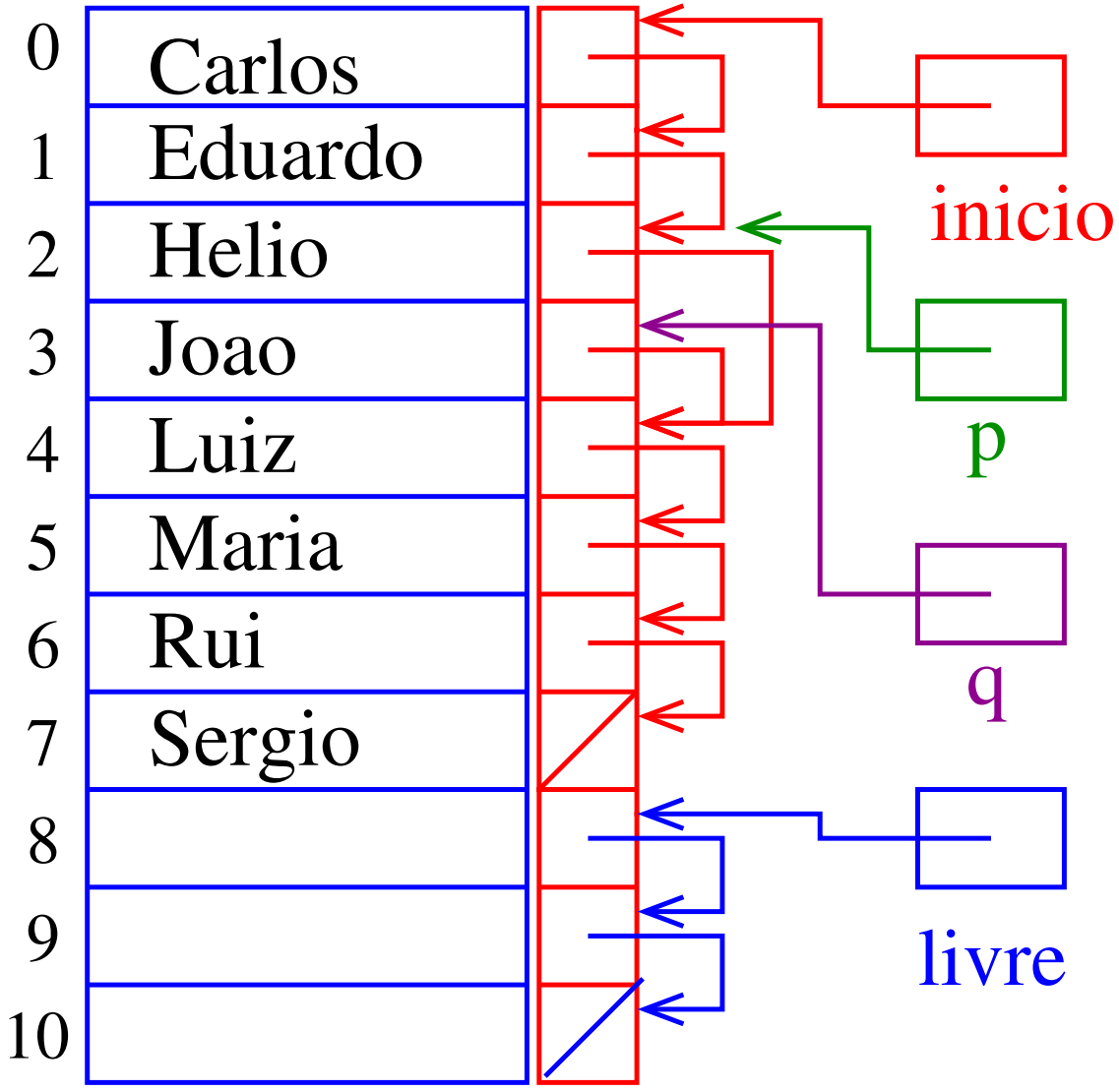
Remover Joao



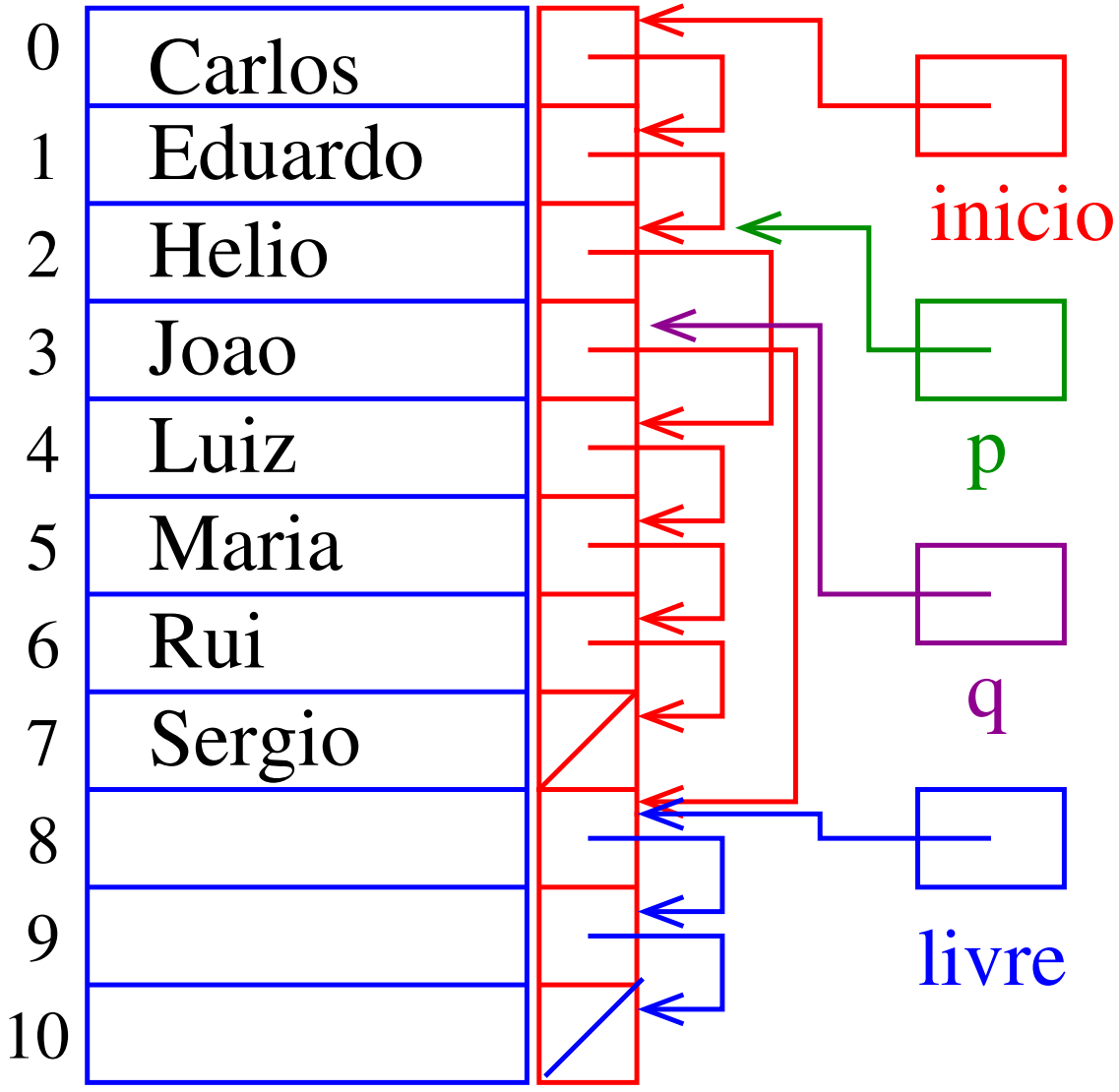
Remover Joao



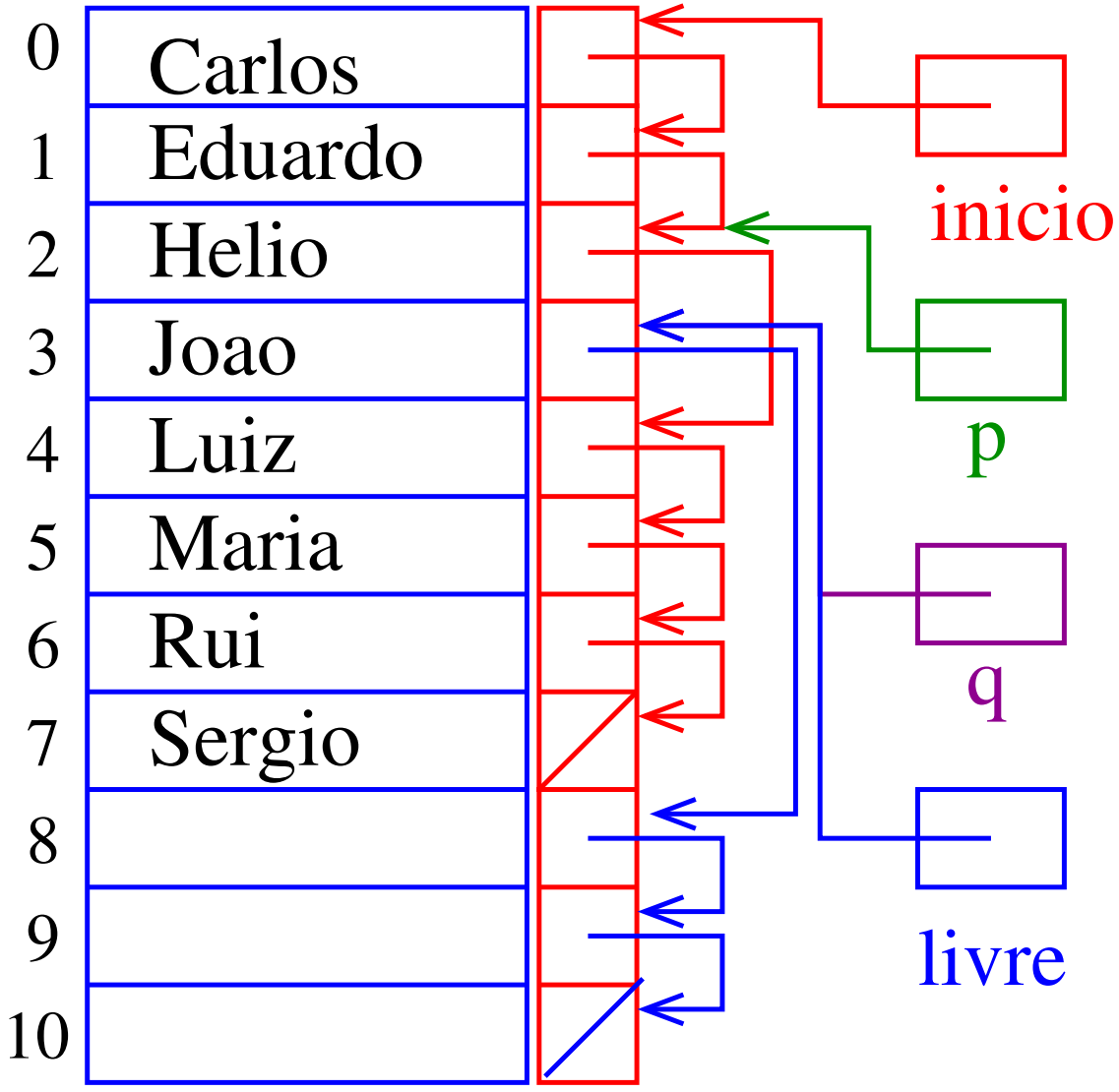
Remover Joao



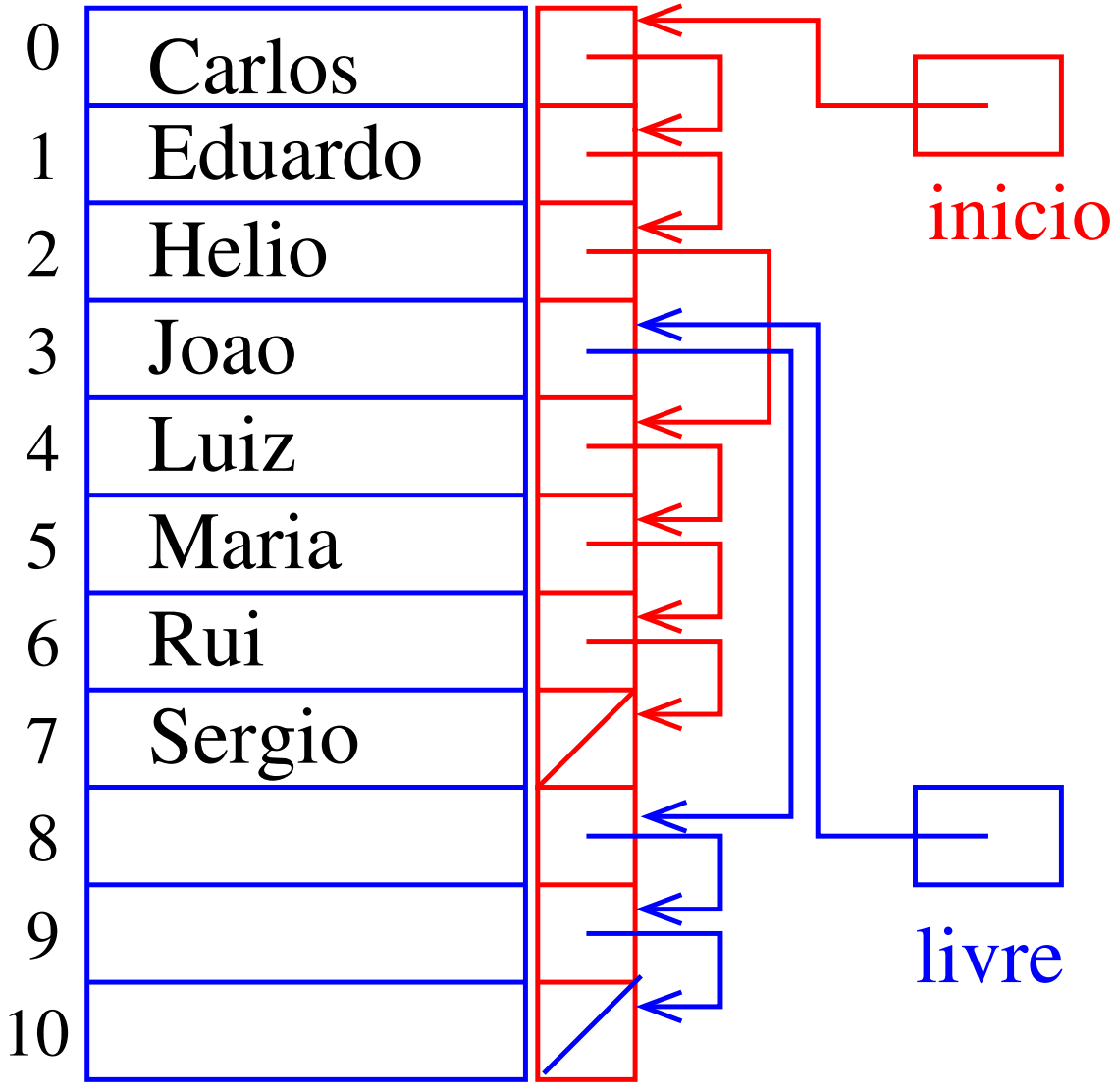
Remover Joao



Remover Joao



Remover Joao



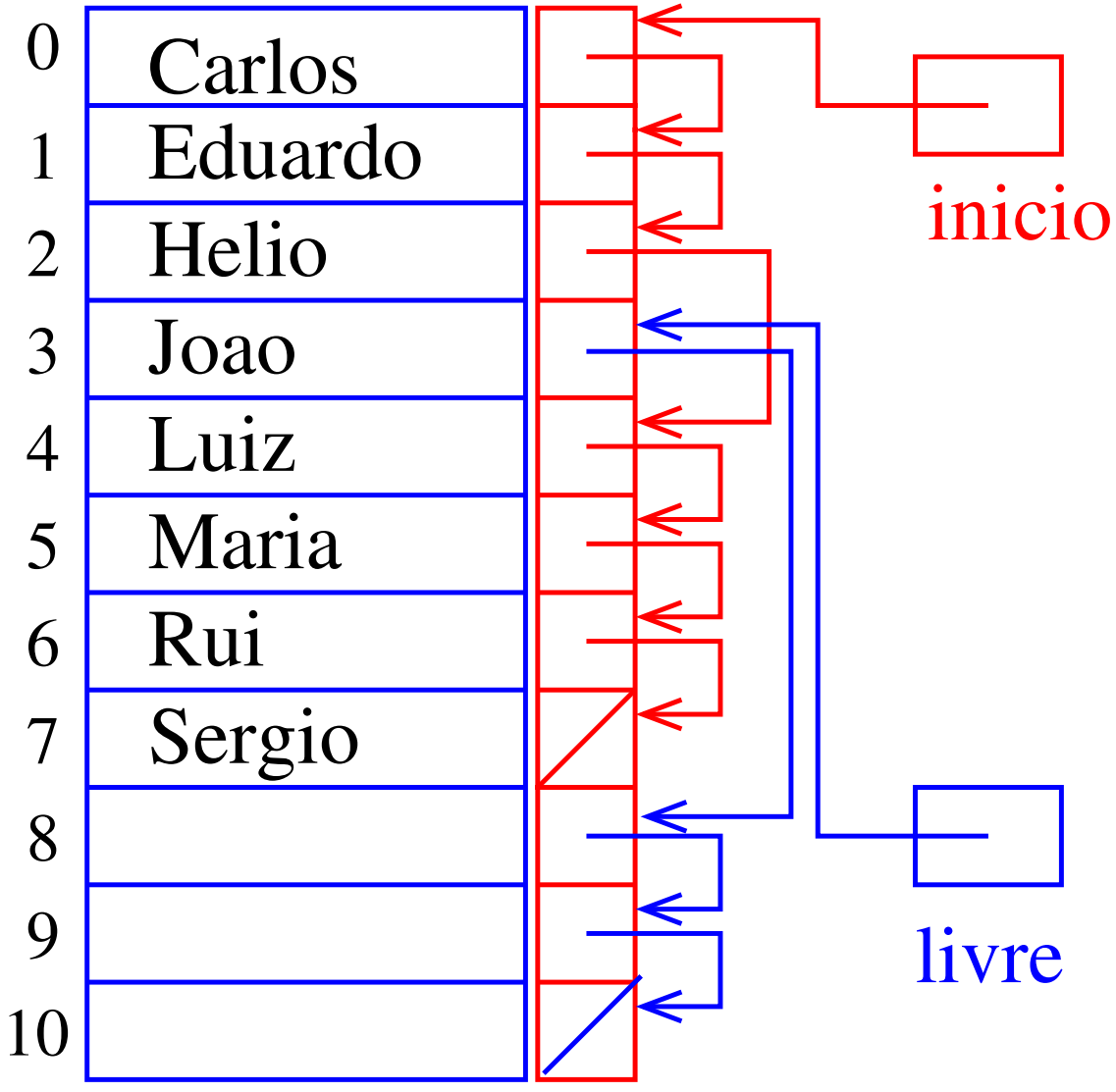
Remoção

Dado um índice `p` de uma célula o trecho de código que remove a célula de índice `v[p].prox` é "essencialmente" o seguinte

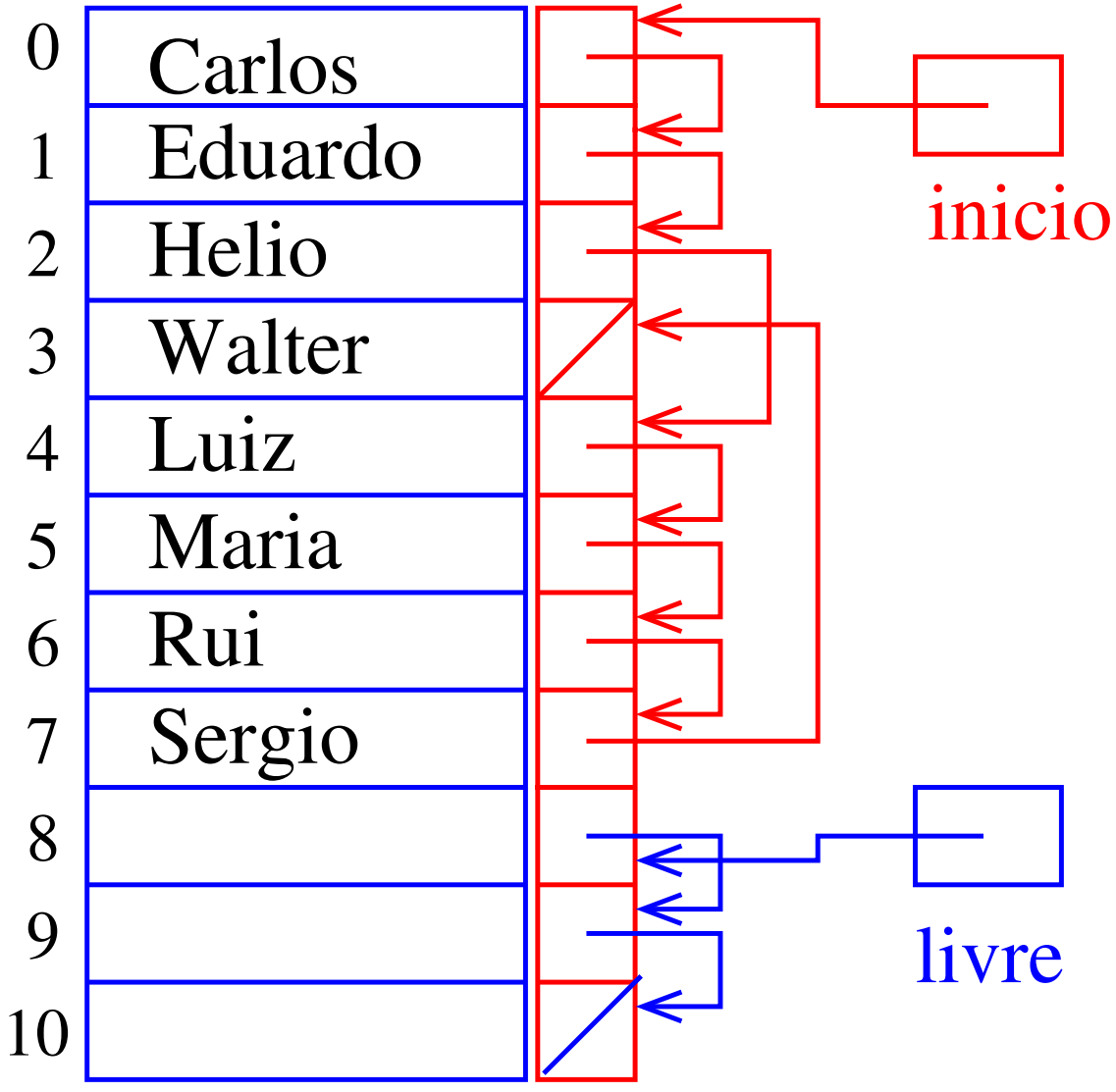
```
q = v[p].prox;
v[p].prox = v[q].prox;
v[q].prox = livre;
livre = q;
```

Dizemos que o consumo de tempo do trecho de código acima é constante, pois **não depende** do tamanho da lista.

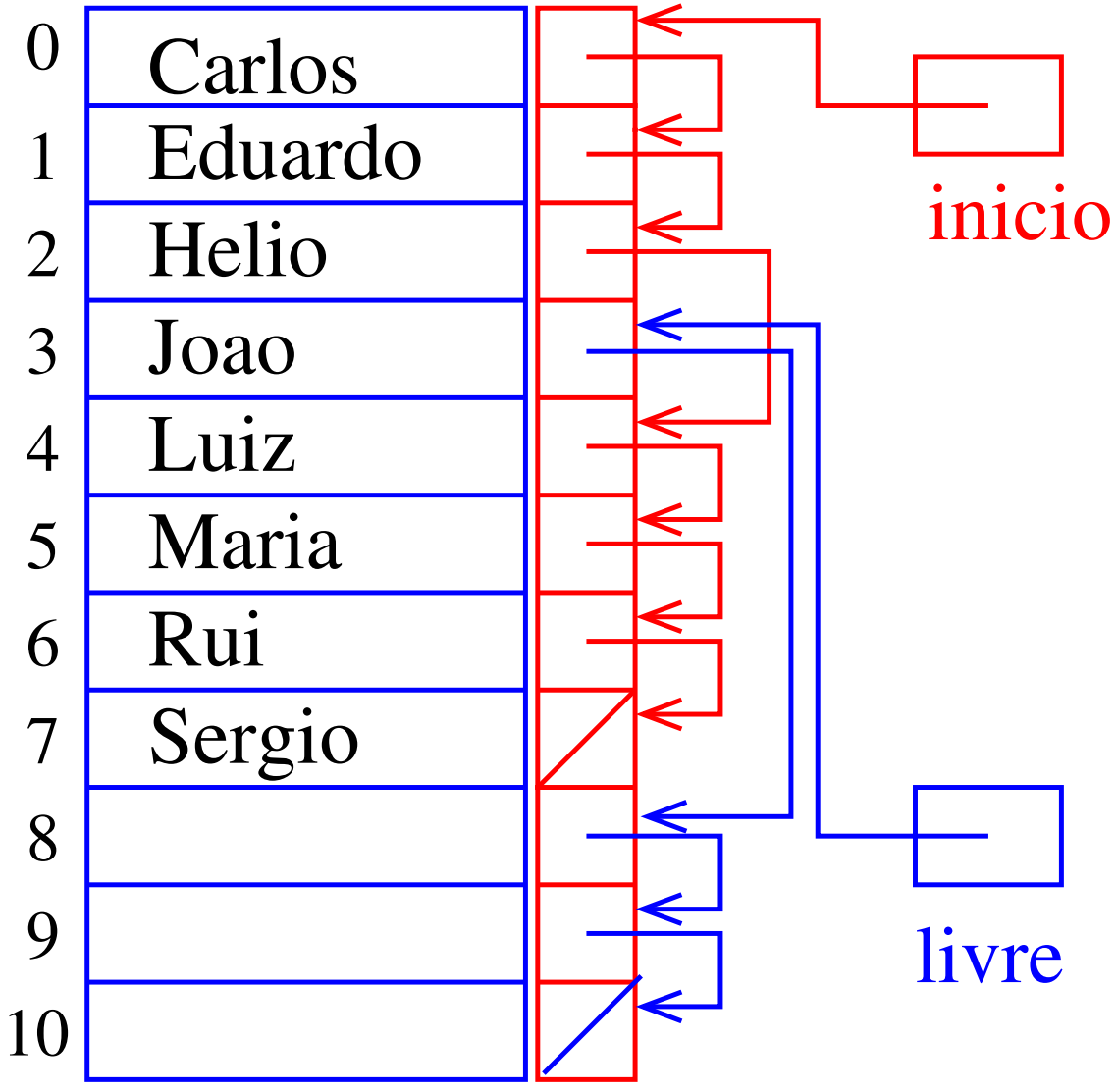
Inserir Walter



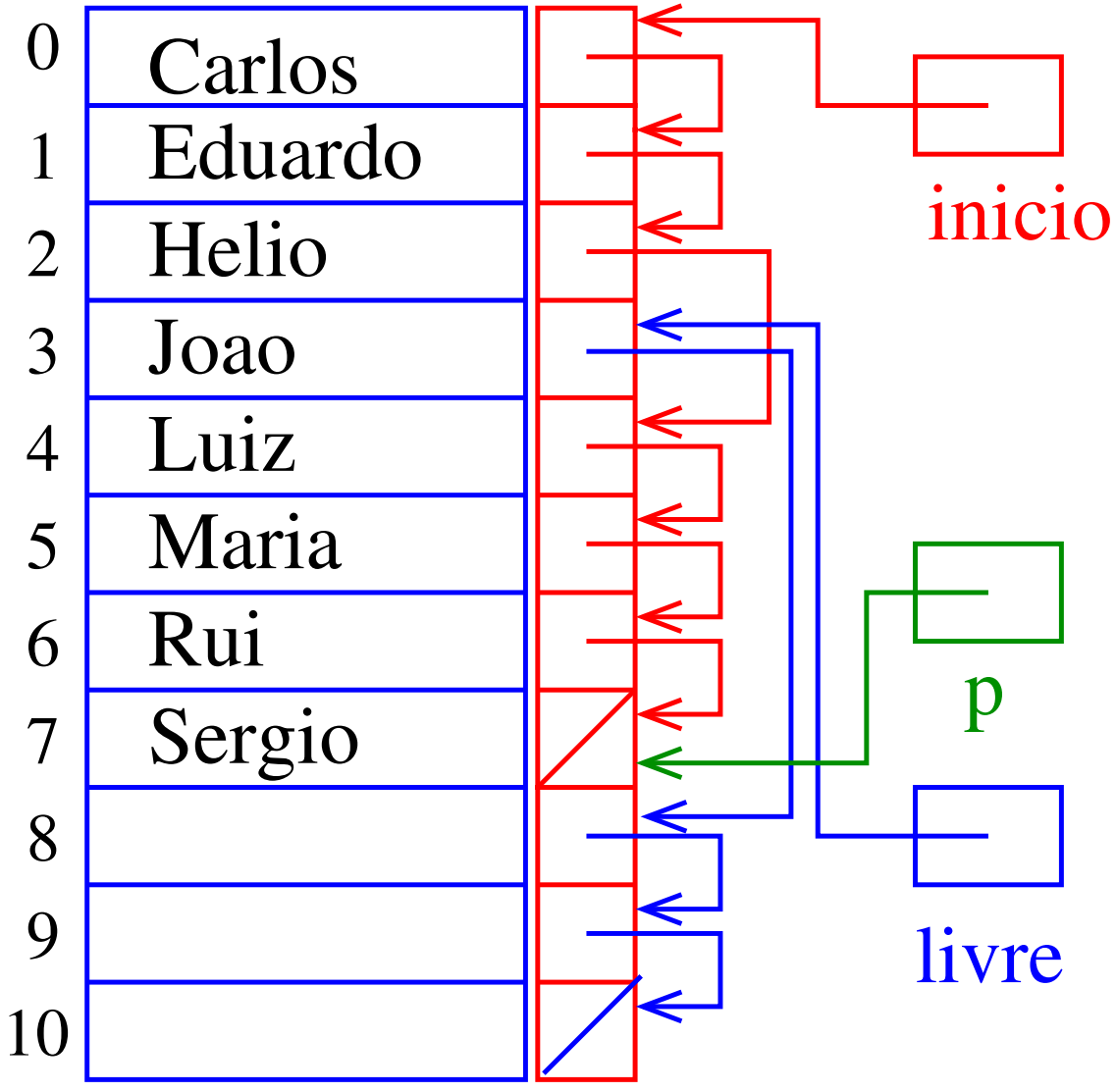
Inserir Walter



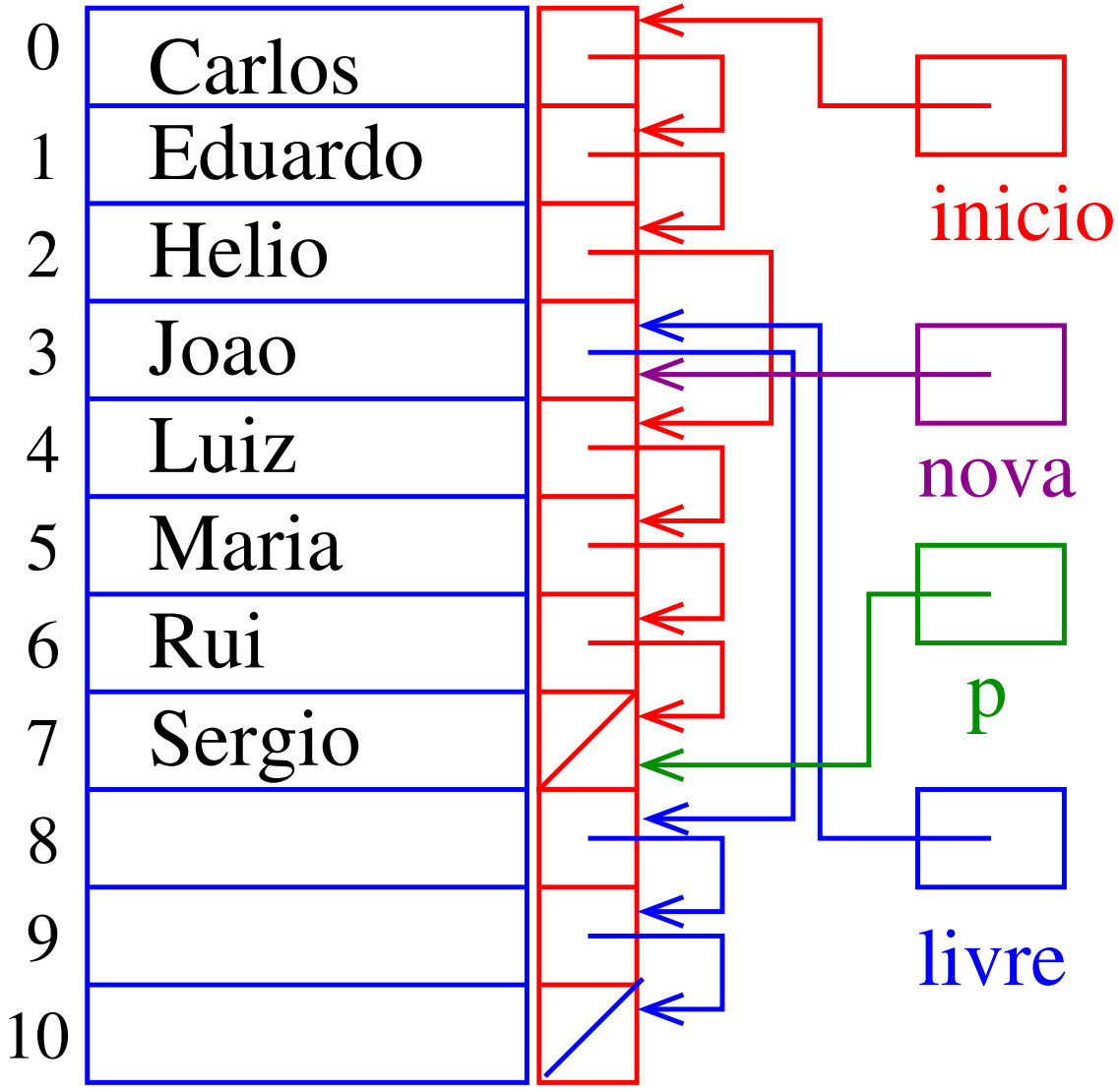
Inserir Walter



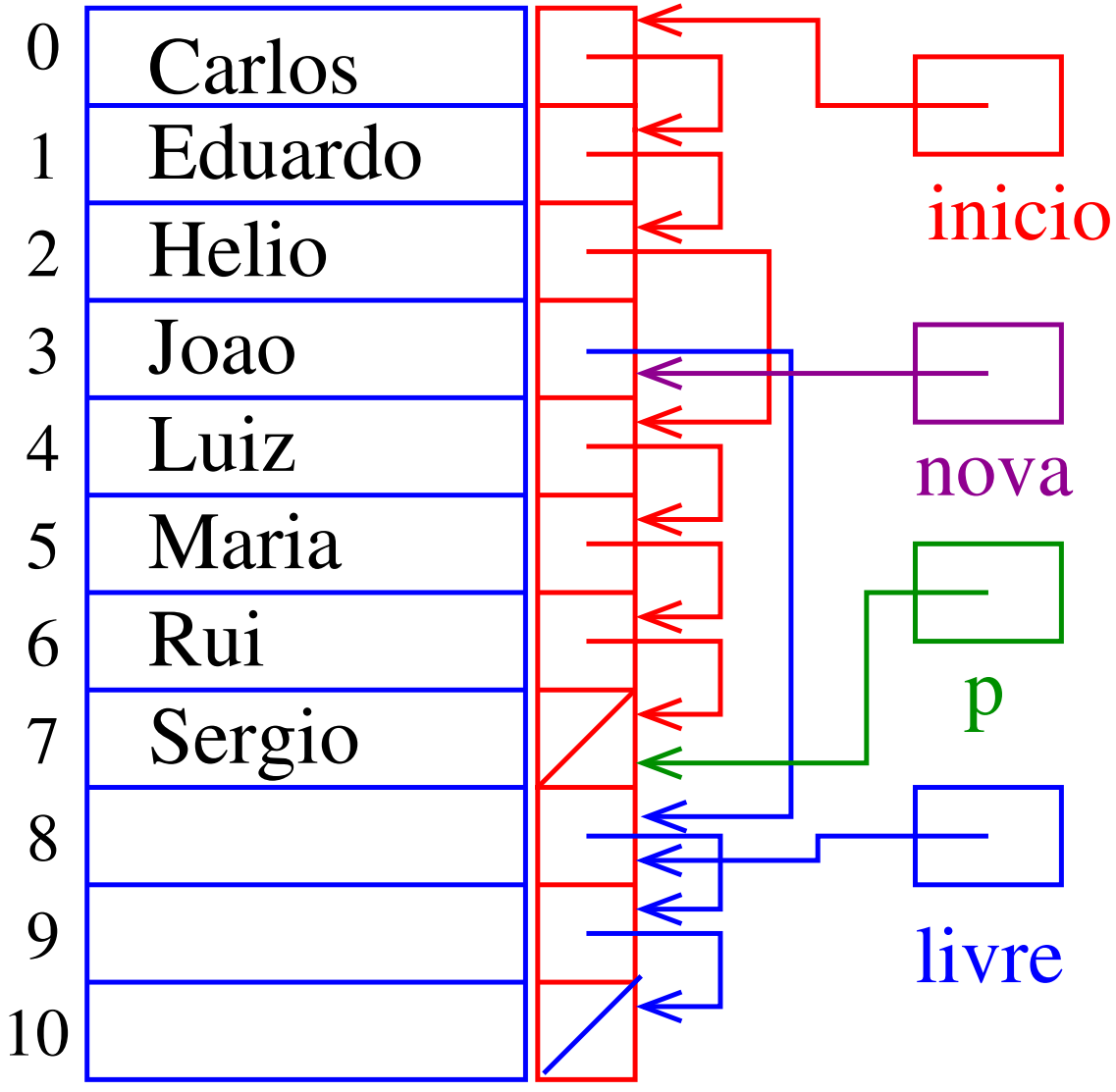
Inserir Walter



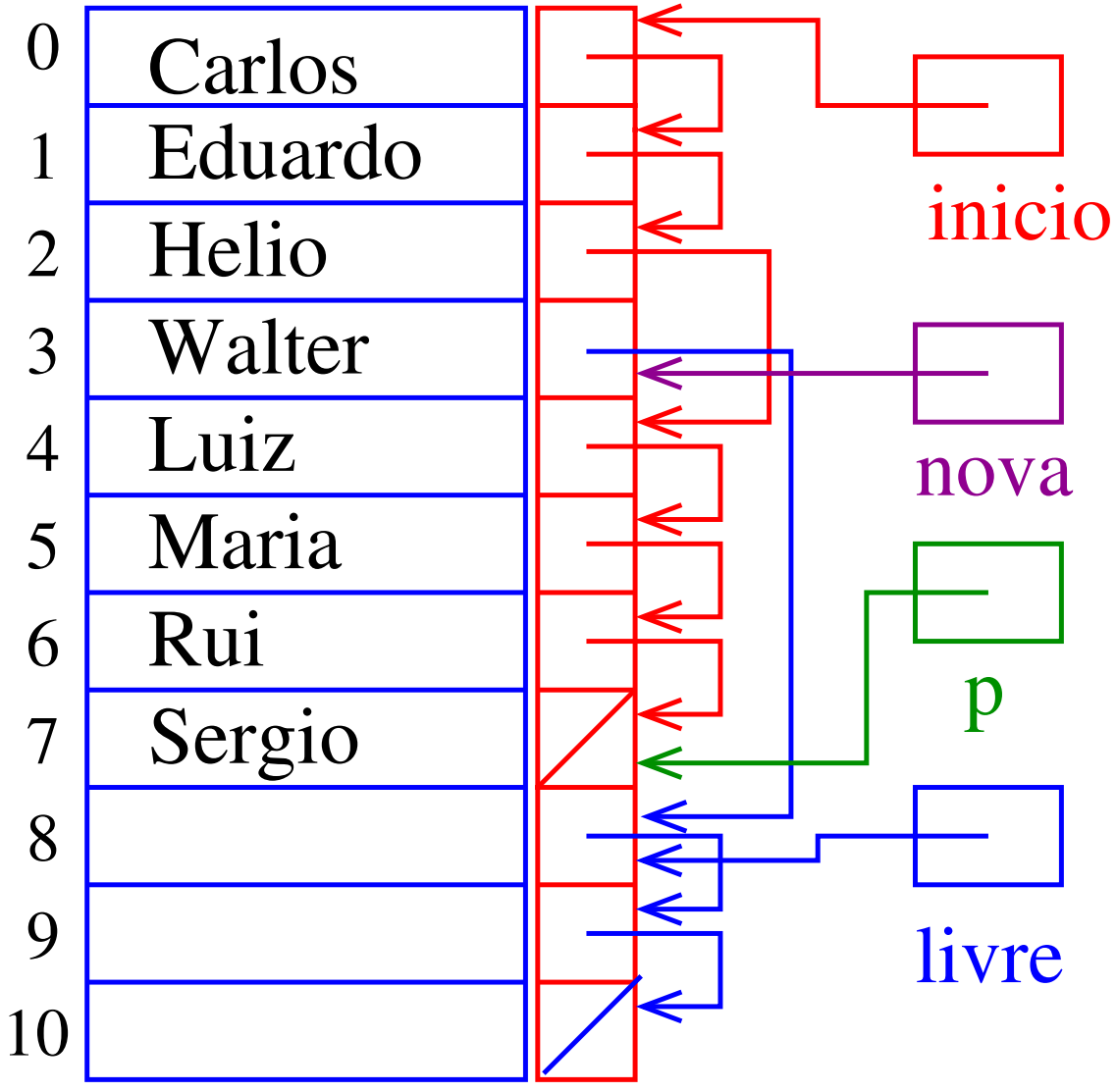
Inserir Walter



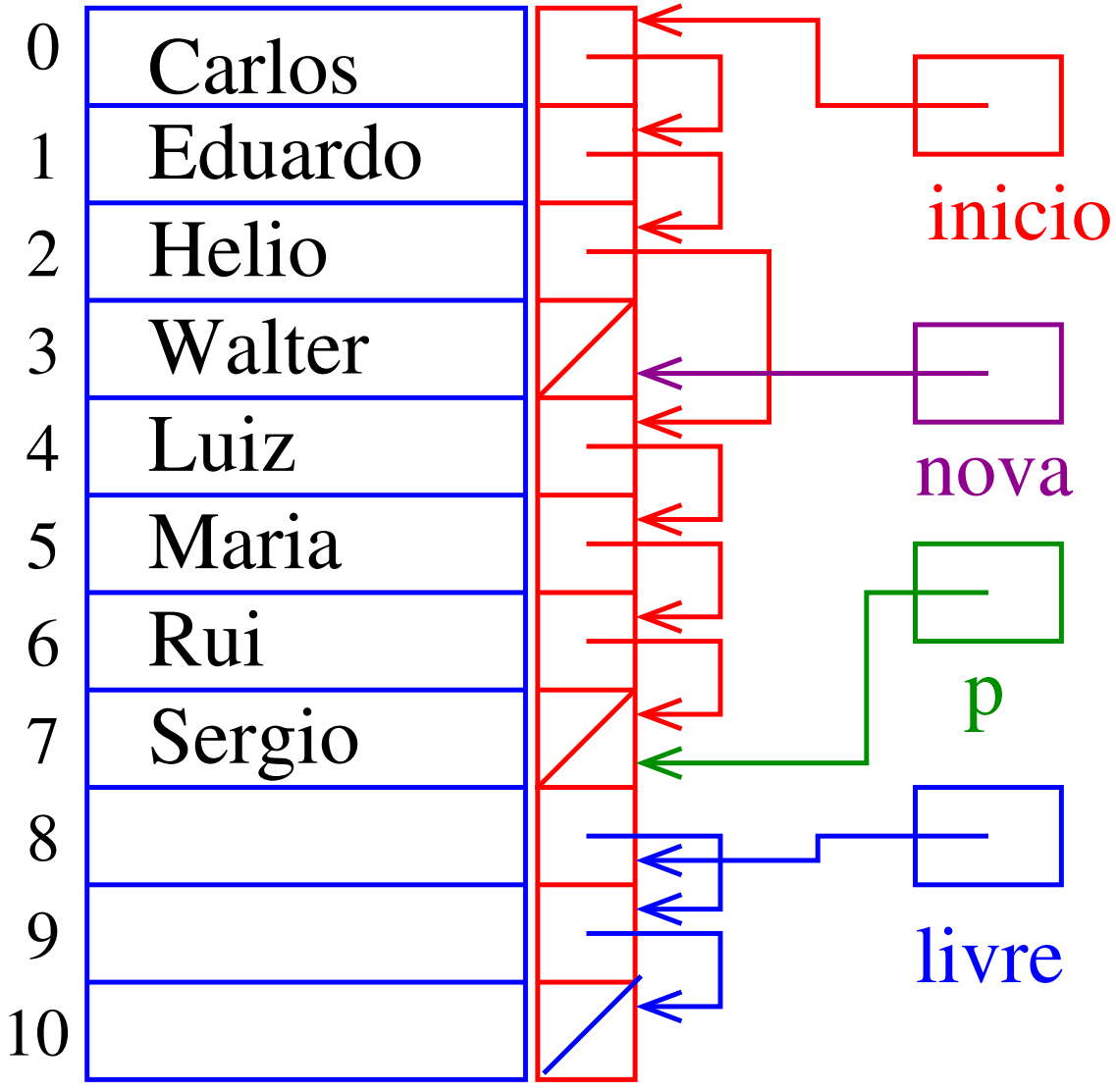
Inserir Walter



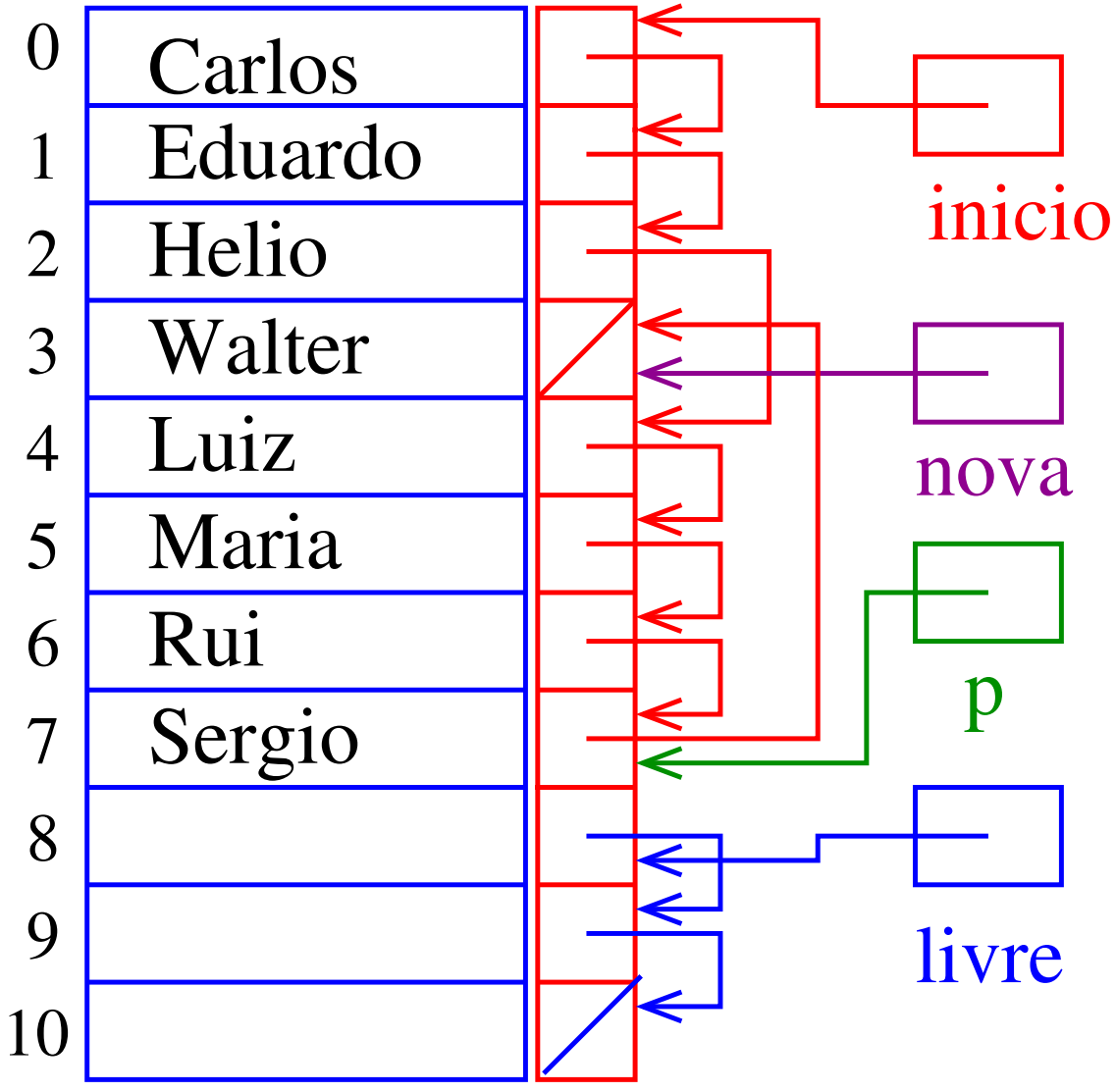
Inserir Walter



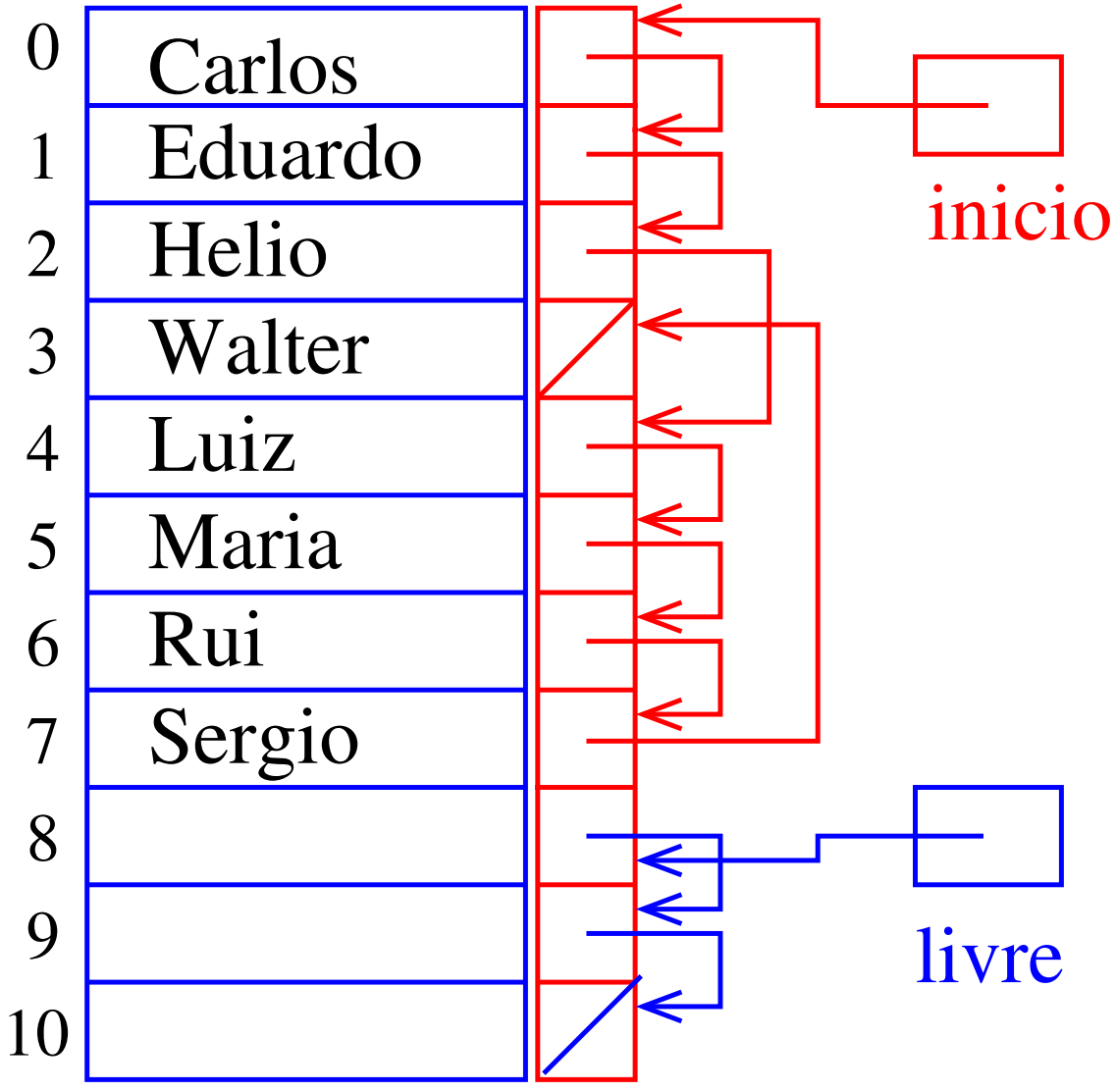
Inserir Walter



Inserir Walter



Inserir Walter



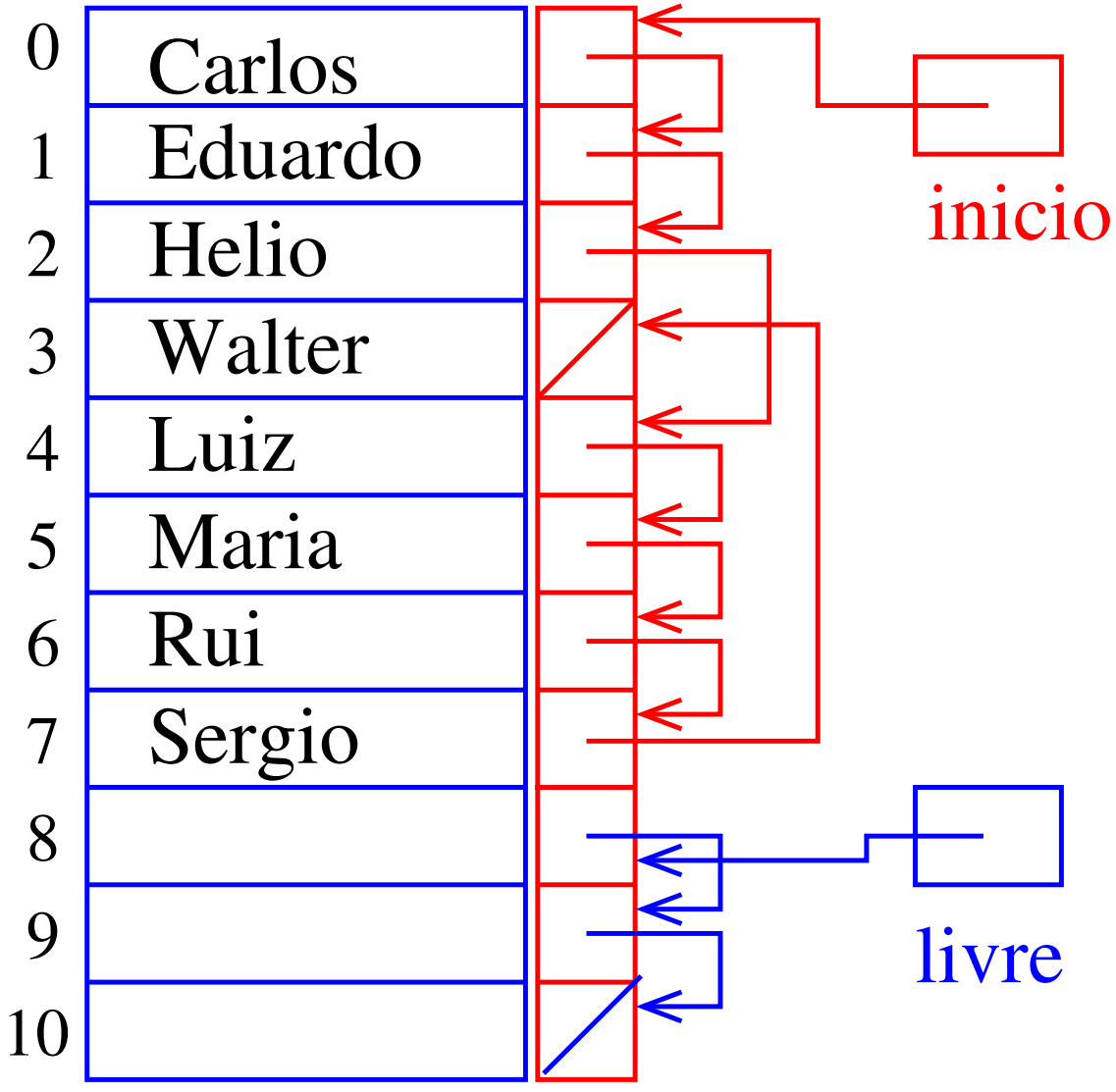
Inserção

O trecho de código para inserir uma nova célula com o elemento `x` entre as células de índice `p` e `v[p].prox` é "essencialmente" o seguinte

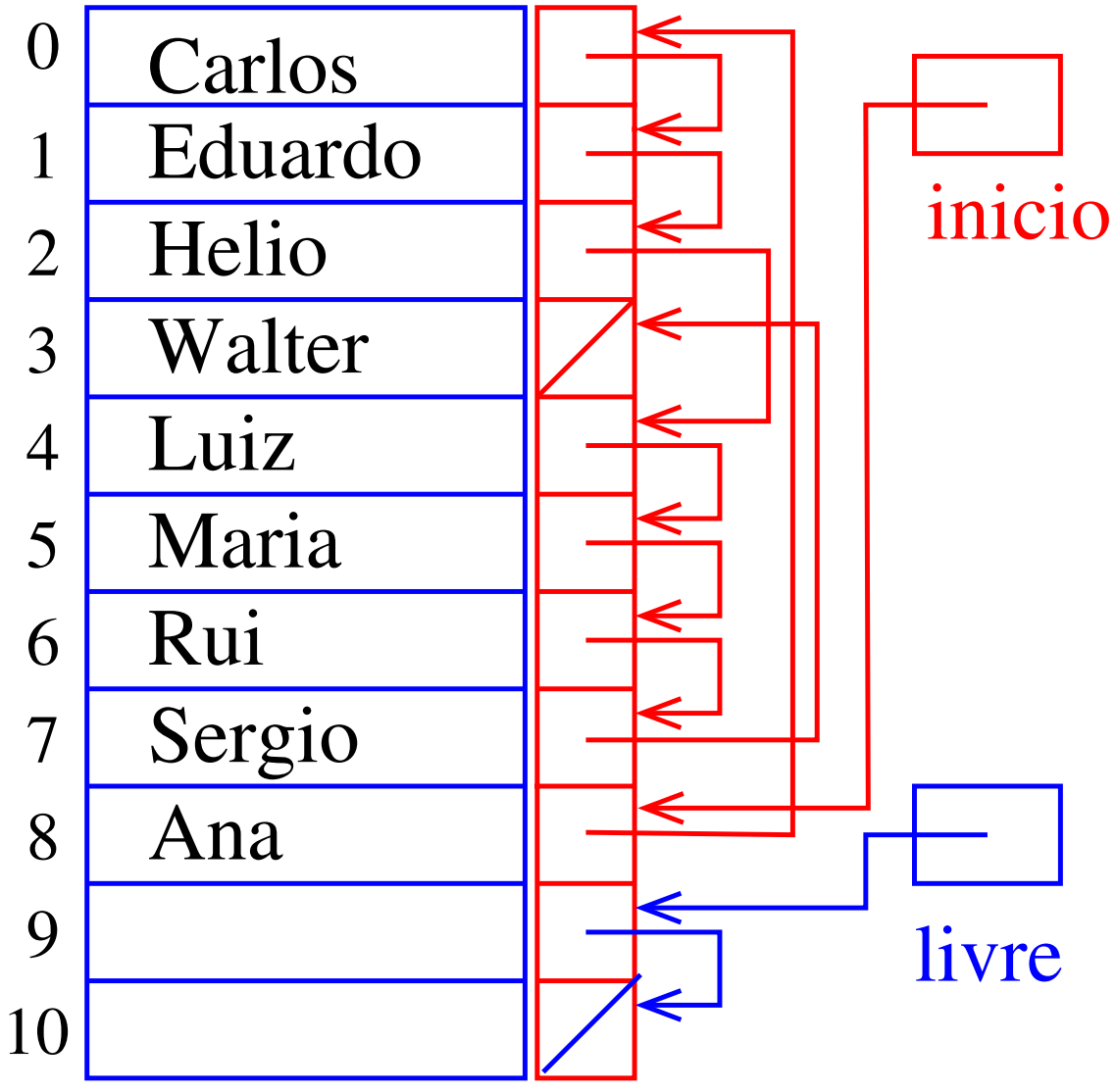
```
nova = livre;
livre = v[livre].prox;
v[nova].conteudo = x;
v[nova].prox = v[p].prox;
v[p].prox = nova;
```

Dizemos que o consumo de tempo do trecho de código acima é constante, pois **não depende** do tamanho da lista.

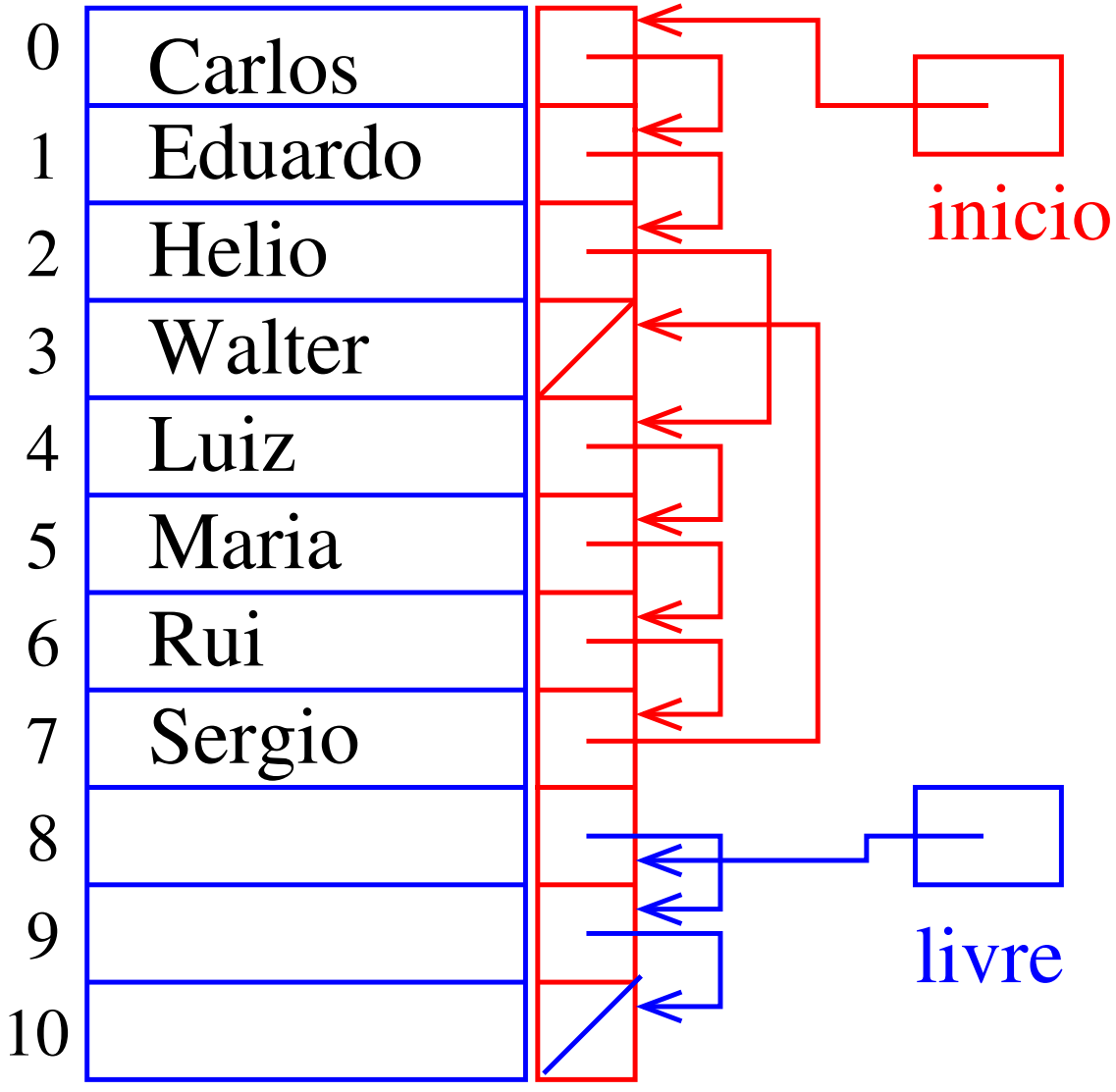
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
9
10
inicio
livre

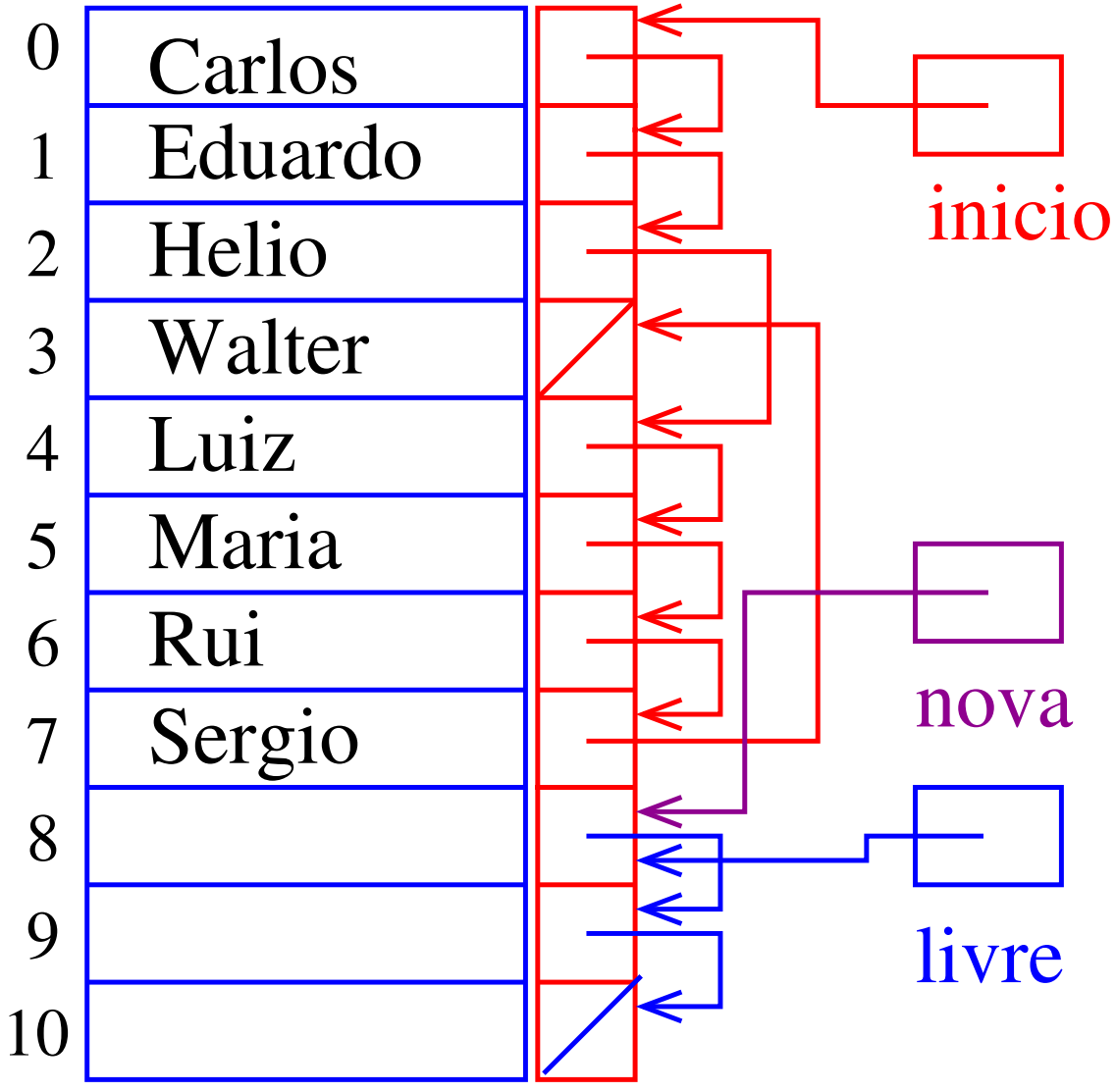
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Ana
9
10
inicio
livre

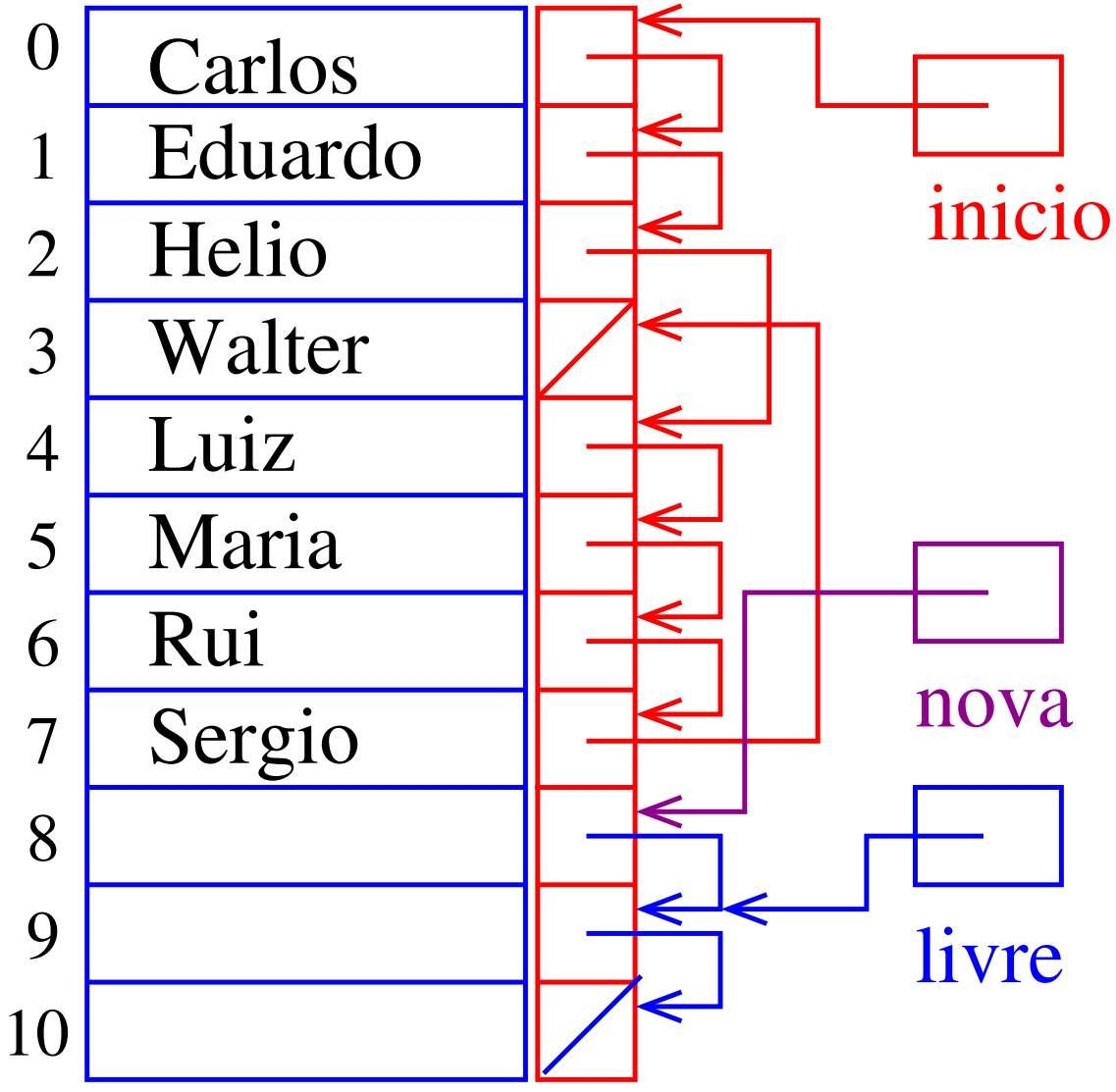
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
9
10
inicio
livre

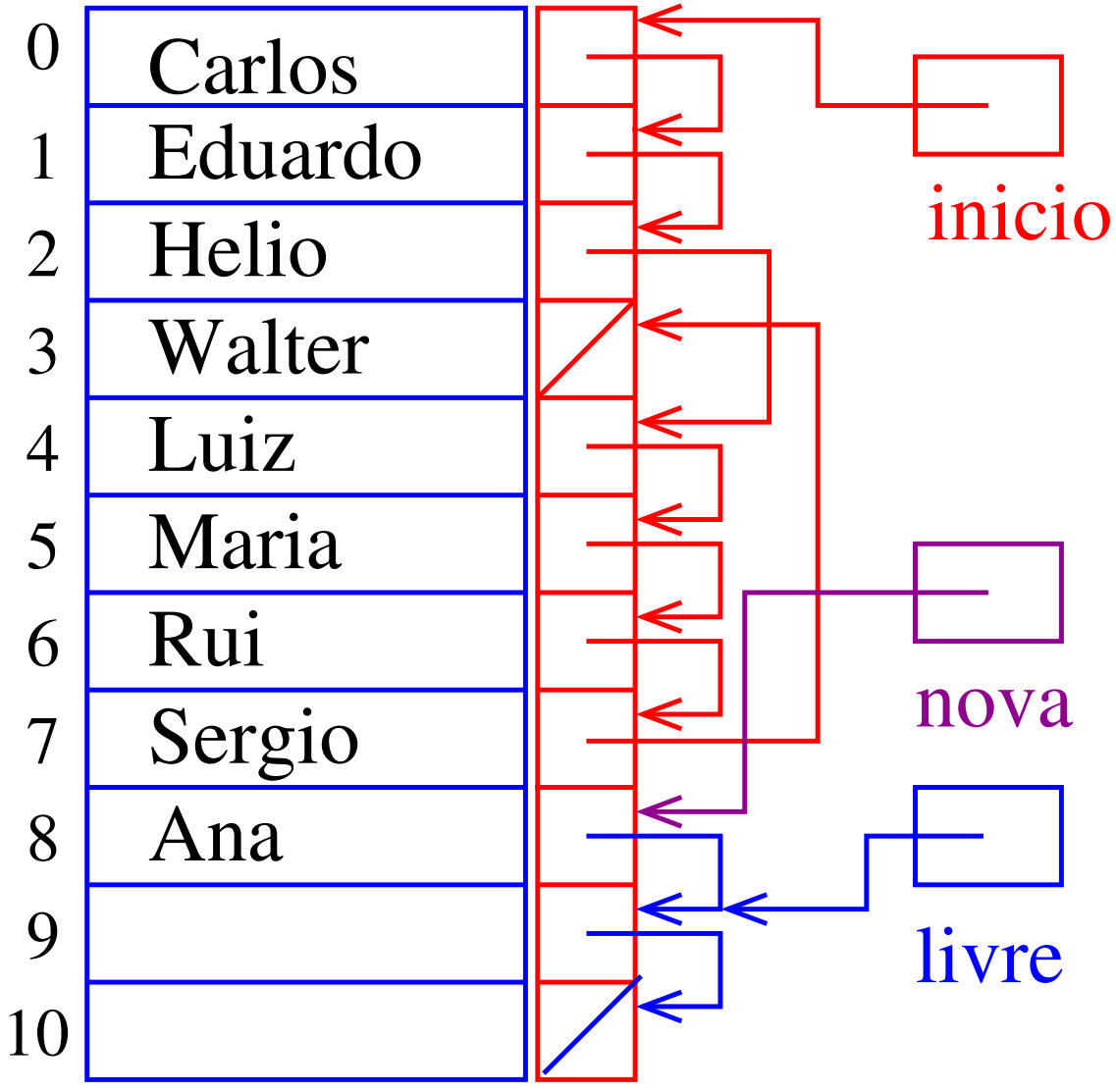
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
9
10
inicio
nova
livre

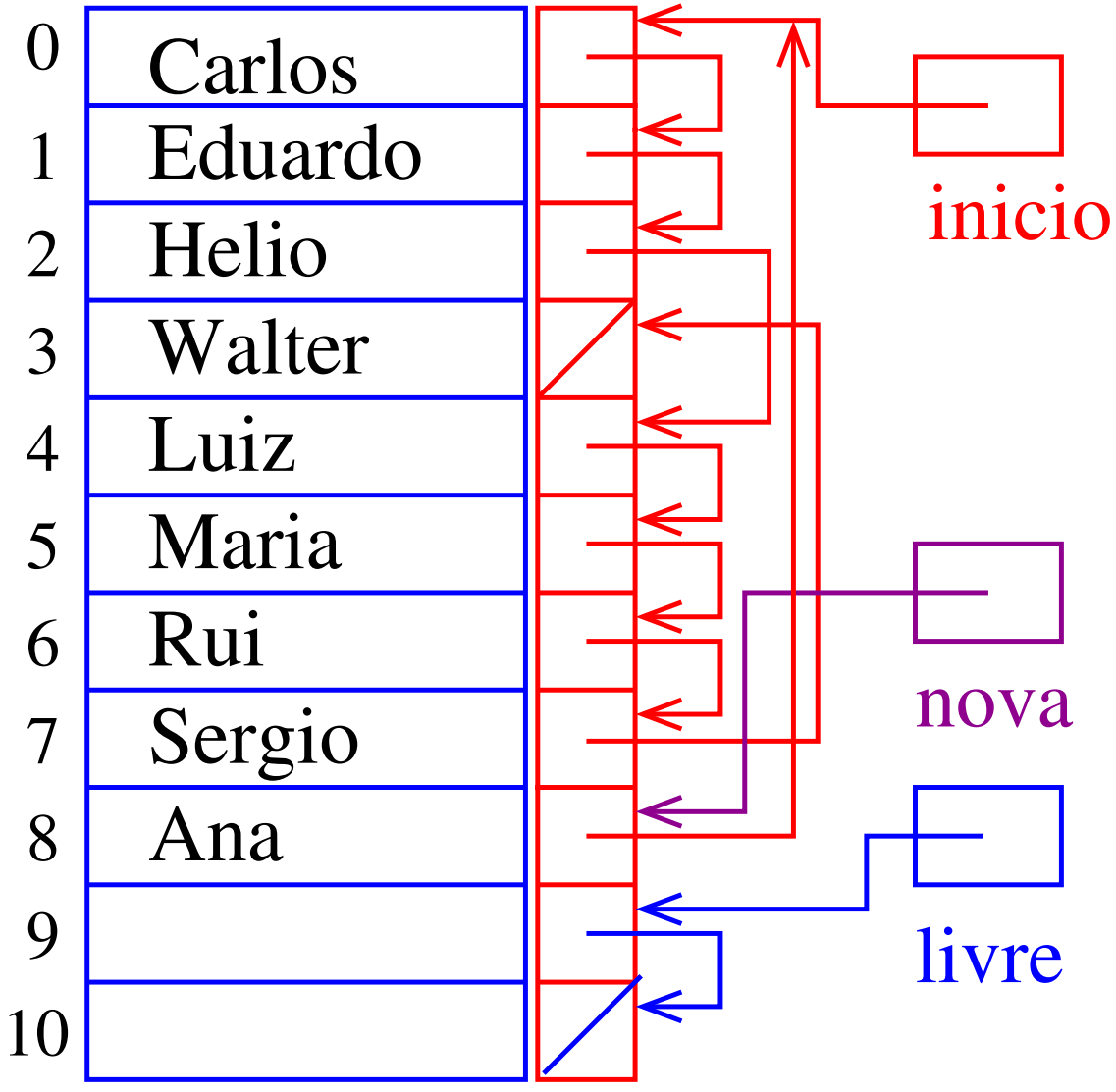
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
9
10
inicio
nova
livre

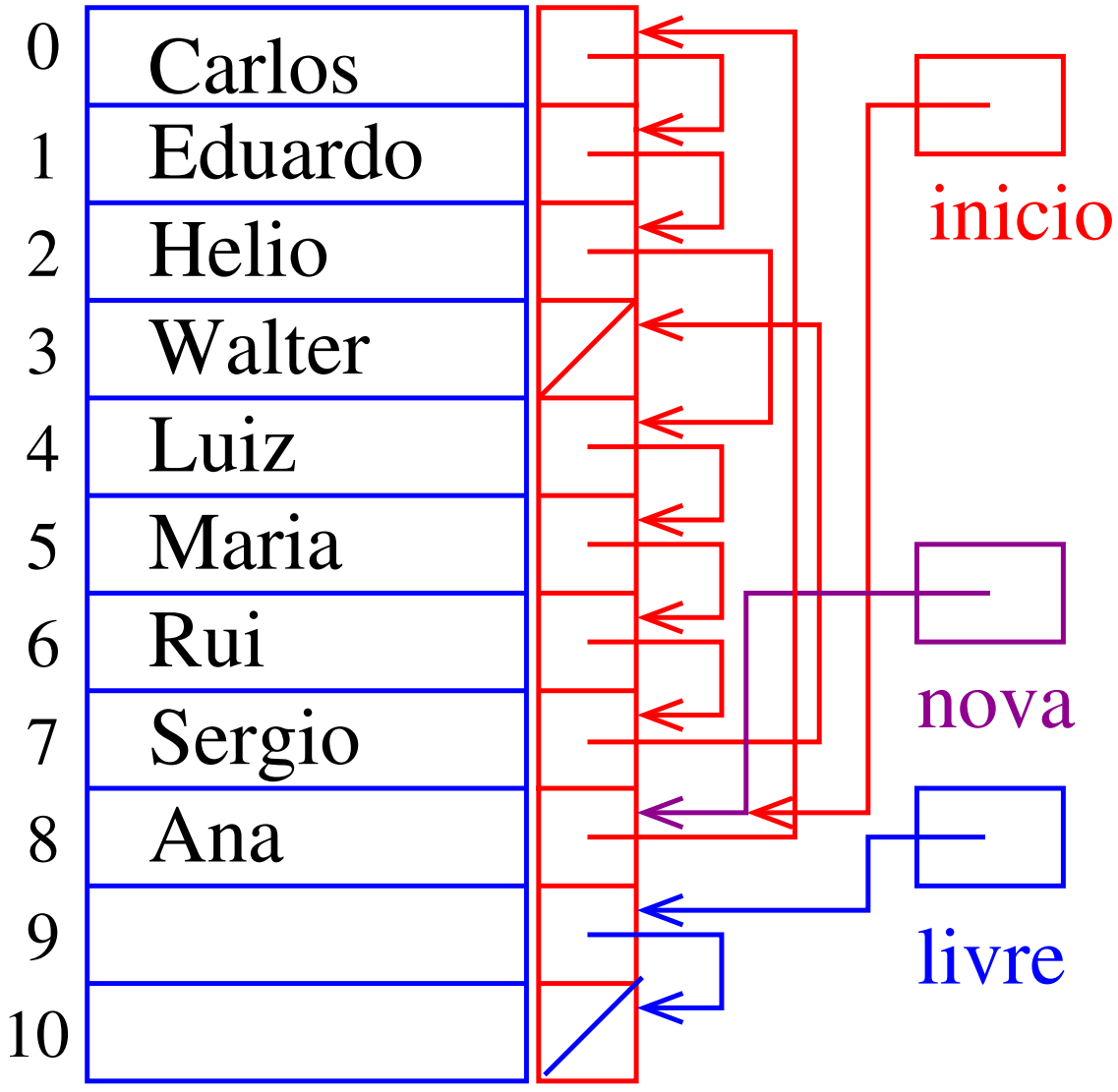
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Ana
9
10
inicio
nova
livre

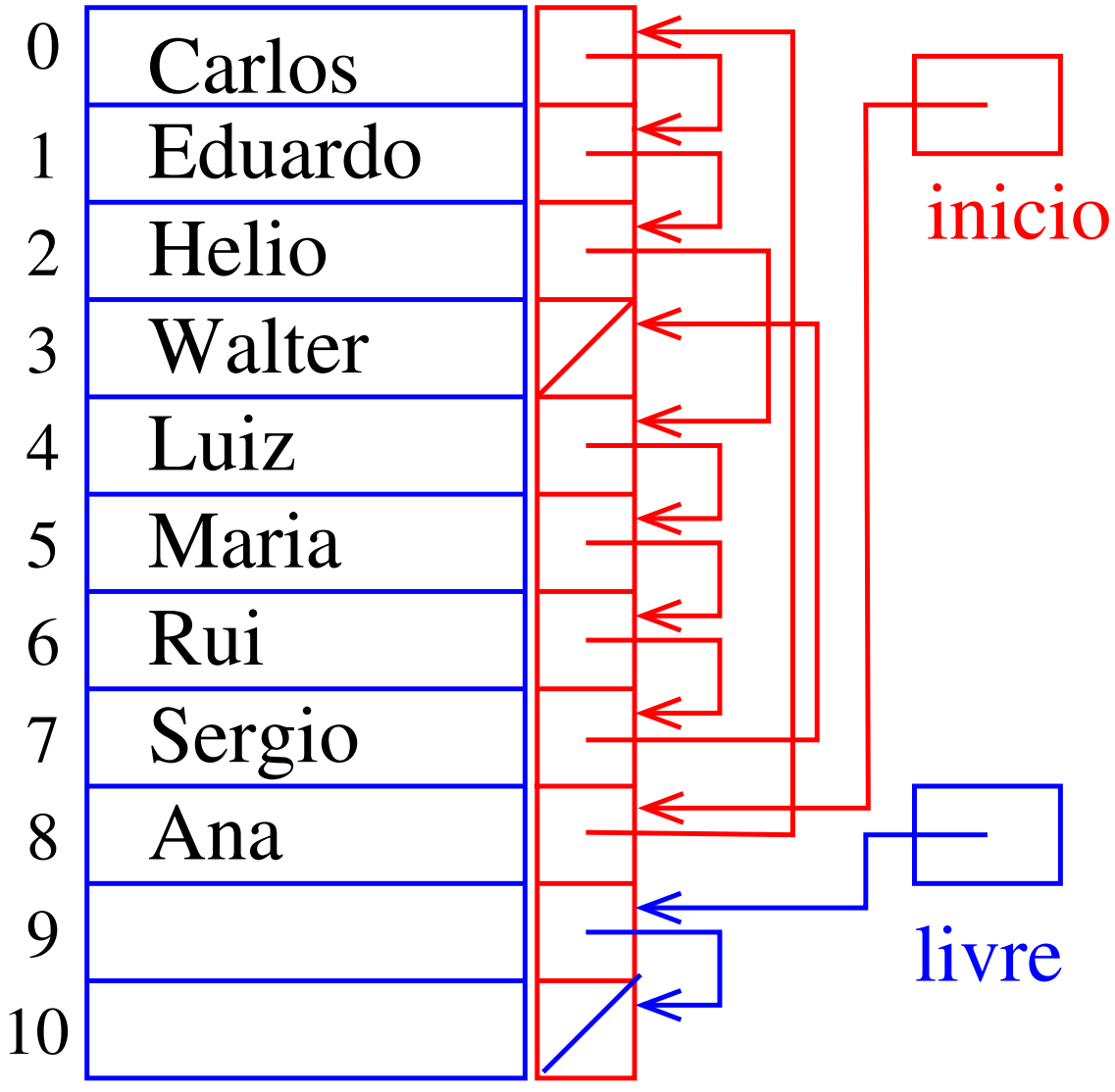
Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Ana
9
10
inicio
nova
livre

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Ana
9
10
inicio
nova
livre

Inserir Ana


0
Carlos
1
Eduardo
2
Helio
3
Walter
4
Luiz
5
Maria
6
Rui
7
Sergio
8
Ana
9
10
inicio
livre

Inserção no início

O trecho de código para inserir uma nova célula com o elemento `x` no início da lista;

```
nova = livre;
livre = v[livre].prox;
v[nova].conteudo = x;
v[nova].prox = inicio;
inicio = nova;
```

Dizemos que o consumo de tempo do trecho de código acima é constante, pois **não depende** do tamanho da lista.