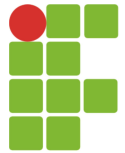
Listas

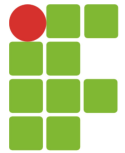
Estruturas de Dados e Algoritmos – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

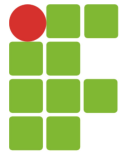
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário



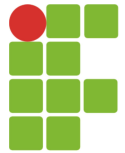
Sumário



Tipo Abstrato de Dados

TAD

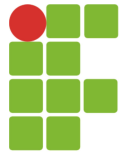
- Um tipo abstrato de dado (TAD) é um modelo matemático para uma classe de estruturas de dados que possuem uma semântica similar.
- Um TAD define as operações essenciais sobre uma estrutura de dados.



Listas

- Lista é um TAD definido como uma sequência de valores em que um determinado valor pode ocorrer múltiplas vezes.
- A lista possui uma cabeça (primeiro elemento da sequência) e uma cauda (último elemento da sequência).
- É interessante que listas possuam operações eficientes na cabeça e na cauda.

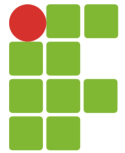
Listas



Listas

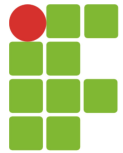
Operações sobre Listas

- Verificar se a lista está vazia;
- Inserção de qualquer posição da lista;
- Inserção na cabeça;
- Inserção na cauda;
- Remoção de qualquer posição da lista;
- Remoção da cabeça da lista;
- Remoção da cauda da lista;
- Acesso à cabeça da lista;
- Acesso à cauda da lista;
- Acesso à qualquer posição da lista;



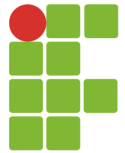
Listas

- Listas podem ser implementadas por vetores ou estruturas dinâmicas auto-referenciadas.
- Nosso foco será em estruturas auto-referenciadas.

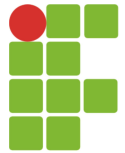


Listas Encadeadas

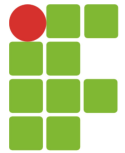
- Uma estrutura auto-referenciado é aquela que contém uma referencia para um elemento do mesmo tipo.
- Em `C`, isto é alcançado através de **ponteiros**.
- Listas Auto-Referenciadas, Listas Encadeadas ou Listas Ligadas!



Sumário

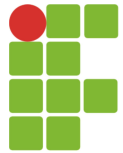


Sumário



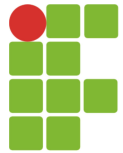
Listas Encadeadas: Definição

```
6  typedef void* (*list_node_constructor_fn) (void*);
7  typedef void (*list_node_destructor_fn)(void *);
```



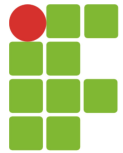
Listas Encadeadas: Definição

```
9  /**
10   @brief list_node_t Definição de nó de lista ligada.
11   O nó de lista ligada contém um ponteiro para um dado genérico (data)
12   e um ponteiro para o próximo nó da lista.
13 **/
14 typedef struct list_node_t{
15     void* data; /*Ponteiro para um dado genérico de qualquer tipo*/
16     struct list_node_t* next; /*ponteiro para o próximo elemento*/
17 }list_node_t;
18
```



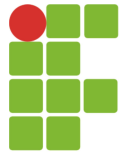
Listas Encadeadas: Definição

```
29  /**
30    @brief list_t Definidção do tipo lista. Contém ponteiros para a cabeça
31    e cauda da lista, as funções construtoras e destrutoras dos elementos
32    das listas e o tamanho da lista.
33  **/
34  typedef struct list_t{
35      list_node_t* head; /*Cabeça da Lista*/
36      list_node_t* tail; /*Cauda da Lista*/
37      list_node_constructor_fn constructor; /*Função para construir o objeto*/
38      list_node_destructor_fn destructor; /*Função para destruir o objeto*/
39      size_t size; /*tamanho da lista*/
40  }list_t;
```



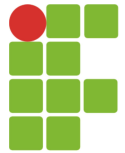
Listas Encadeadas: Definição

```
20 /**
21  * @brief list_iterator_t Tipo para iterar na lista ligada.
22  * O list_iterator_t é apenas um apelido para list_node_t*. Ele é utilizado
23  * no código quando o propósito é andar na lista. Deixa o código mais claro,
24  * pois quando declarado indica explicitamente a intenção da variável.
25  */
26 typedef list_node_t* list_iterator_t;
```



Listas Encadeadas

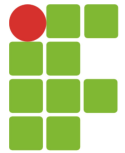
- Em comparação com a implementação em vetores, listas encadeadas possuem vantagens e desvantagens.



Listas Encadeadas

Vantagens

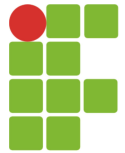
- Estrutura dinâmica: pode aumentar facilmente.
- Inserção em qualquer posição da lista não ocasiona um deslocamento dos elementos posteriores.
- Permite utilizar regiões não contíguas de memória.
- Gerência de simples.



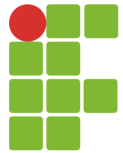
Listas Encadeadas

Desvantagens

- Espaço extra para armazenar ponteiros (implícitos em vetores).
- Não possui acesso aleatório em tempo constante.

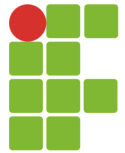


Sumário

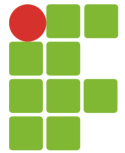


Listas Encadeadas: Inicialização

```
45  /** Inicialização dos membros de uma lista **/
46  void list_initialize(list_t** l,list_node_constructor_fn constructor,
47      list_node_destructor_fn destructor){
48      /** Aloca espaço para a estrutura lista **/
49      (*l) = mallocx(sizeof(list_t));
50      /** Cabeça aponta para NULL**/
51      (*l)->head = NULL;
52      /** Cauda aponta para NULL**/
53      (*l)->tail = NULL;
54      /** Tamanho de uma lista recém inicializada é 0 **/
55      (*l)->size = 0;
56      /** Atribuição da função construtora **/
57      (*l)->constructor = constructor;
58      /** Atribuição da função destrutora **/
59      (*l)->destructor = destructor;
60  }
```

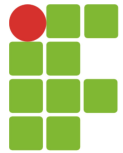


Sumário



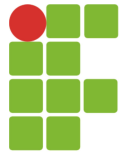
Listas Encadeadas: Funções Auxiliares

```
273 /** Retorna o tamanho da lista **/
274 size_t list_size(list_t* l){
275     return l->size;
276 }
```



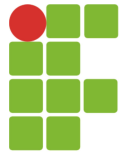
Listas Encadeadas: Funções Auxiliares

```
278 /** Retorna verdadeiro se a lista está vazia, e falso caso contrário **/
279 size_t list_empty(list_t* l){
280     return list_size(l)==0 ? 1 : 0;
281 }
```



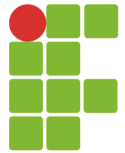
Listas Encadeadas: Funções Auxiliares

```
25  /** Cria o nó de uma lista **/
26  static list_node_t* list_new_node(void* data,list_node_constructor_fn constructor){
27      /** aloca espaço para novo nó **/
28      list_node_t* new_node =  mallocx(sizeof(list_node_t));
29      /** Constrói o novo dado através da função construtora **/
30      new_node->data = constructor(data);
31      /** Atribui o ponteiro para o próximo como NULL **/
32      new_node->next = NULL;
33      /** Retorna o nó alocado **/
34      return new_node;
35  }
```

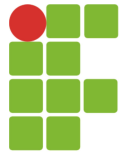


Listas Encadeadas: Funções Auxiliares

```
37  /** Deleta o nó de uma lista **/
38  static void list_delete_node(list_node_t* n,list_node_destructor_fn destructor){
39      /** Chama a função destrutora para o dado do nó **/
40      destructor(n->data);
41      /** Libera o nó **/
42      free(n);
43  }
```



Sumário

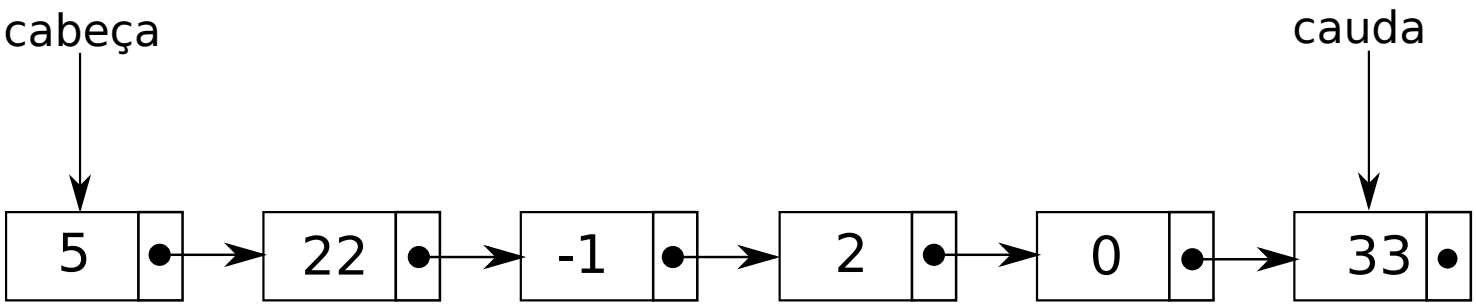


Listas Encadeadas: Inserção

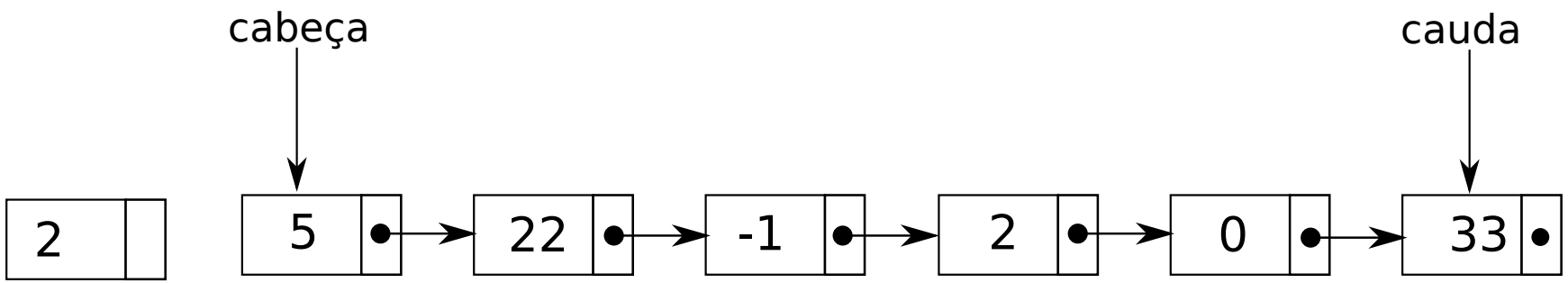
Inserção

- Inserções na cabeça e na cauda da lista, podem ser efetuadas em $\Theta(1)$.
- Inserções em posições aleatórias, requerem acesso sequencial na lista, e portanto tempo $\Theta(n)$.

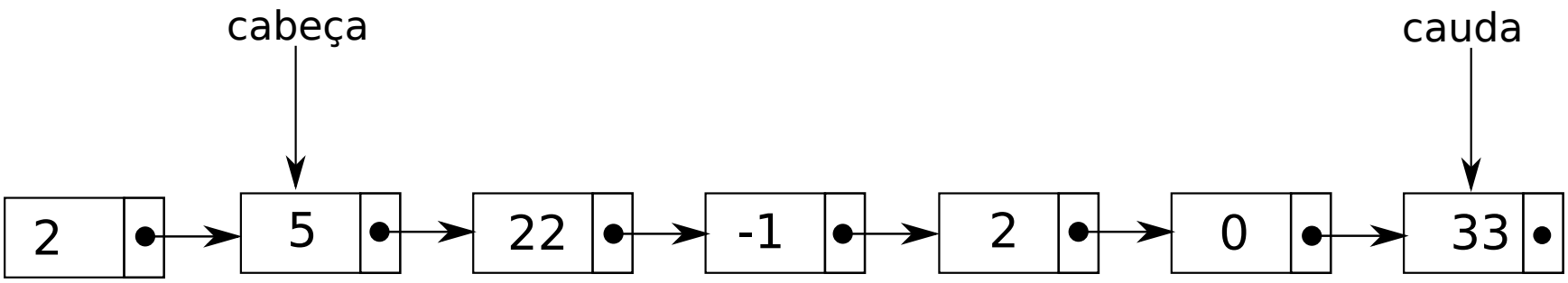
Listas Encadeadas: Inserção na Cabeça



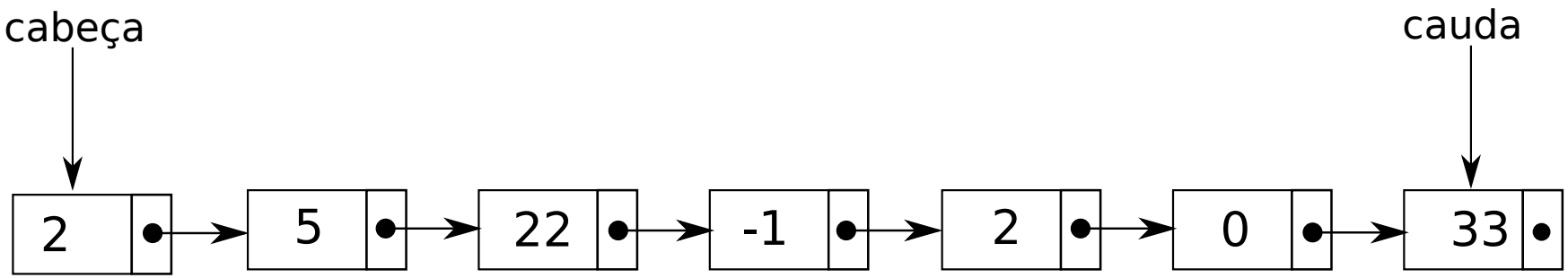
Listas Encadeadas: Inserção na Cabeça

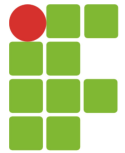


Listas Encadeadas: Inserção na Cabeça



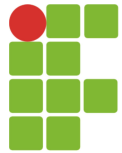
Listas Encadeadas: Inserção na Cabeça



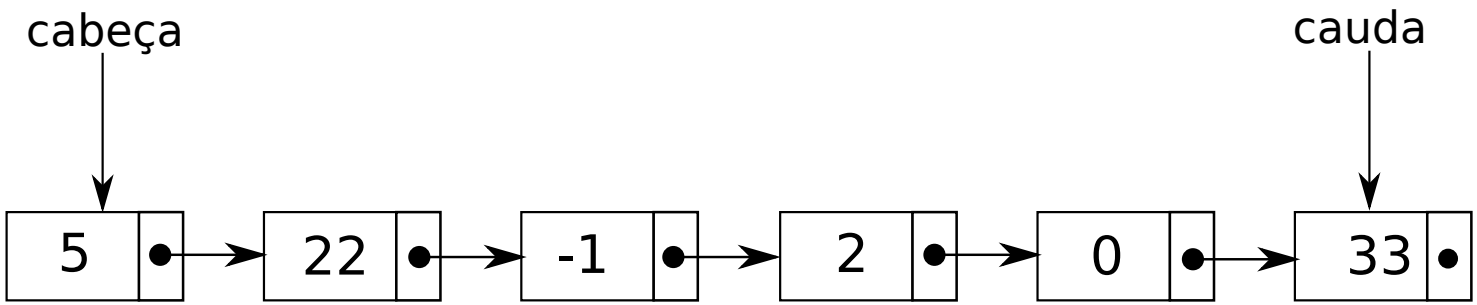


Listas Encadeadas: Inserção na Cabeça

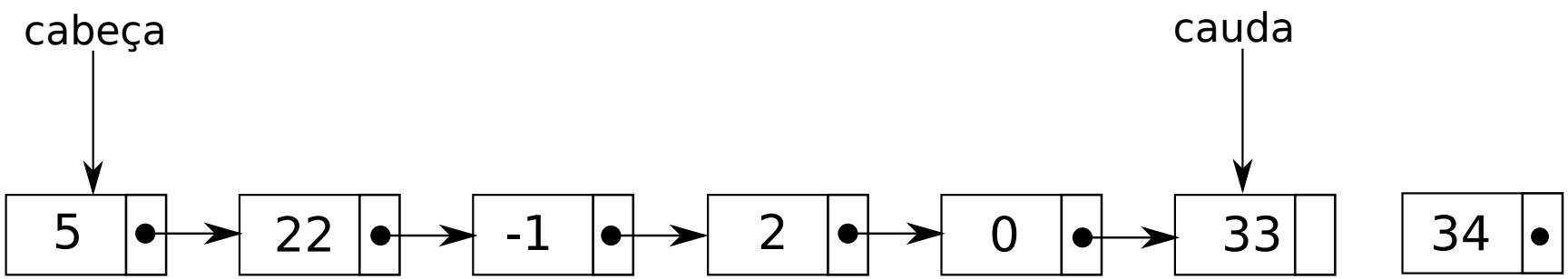
```
110 /** Insere um elemento na cabeça da lista **/
111 void list_prepend(list_t* l,void* data){
112     /** Cria um novo nó ao invocar list_new_node **/
113     list_node_t* new_node = list_new_node(data,l->constructor);
114     /** Novo nó estabelece uma ligação para a cabeça antiga **/
115     new_node->next = l->head;
116     /** Cabeça antiga aponta agora para o nó recém criado **/
117     l->head = new_node;
118     /** Se a lista estava vazia, a cauda também deve apontar para o nó recém
119      * criado **/
120     if(list_empty(l)){
121         l->tail = new_node;
122     }
123     /** O tamanho da lista é incrementado **/
124     l->size++;
125 }
```



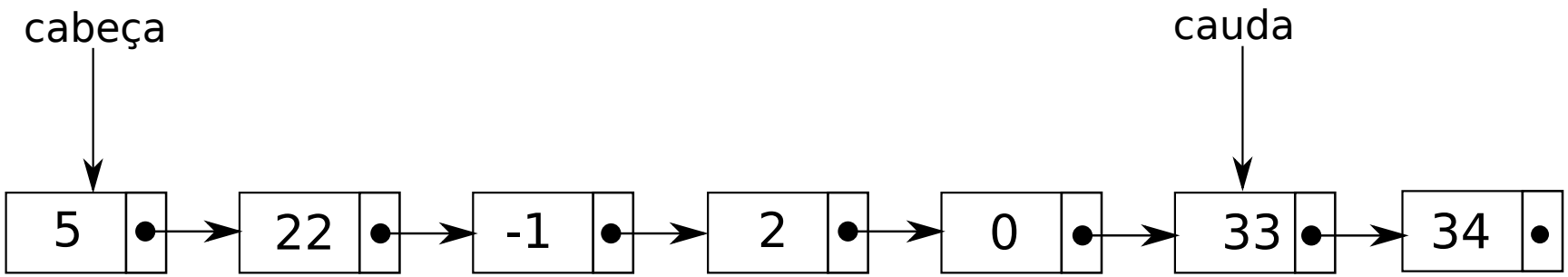
Listas Encadeadas: Inserção na Cauda

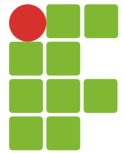


Listas Encadeadas: Inserção na Cauda

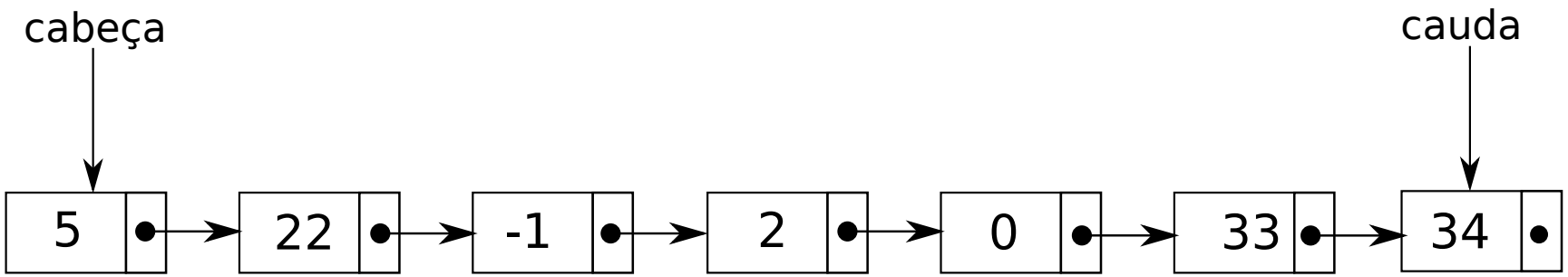


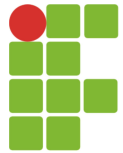
Listas Encadeadas: Inserção na Cauda





Listas Encadeadas: Inserção na Cauda

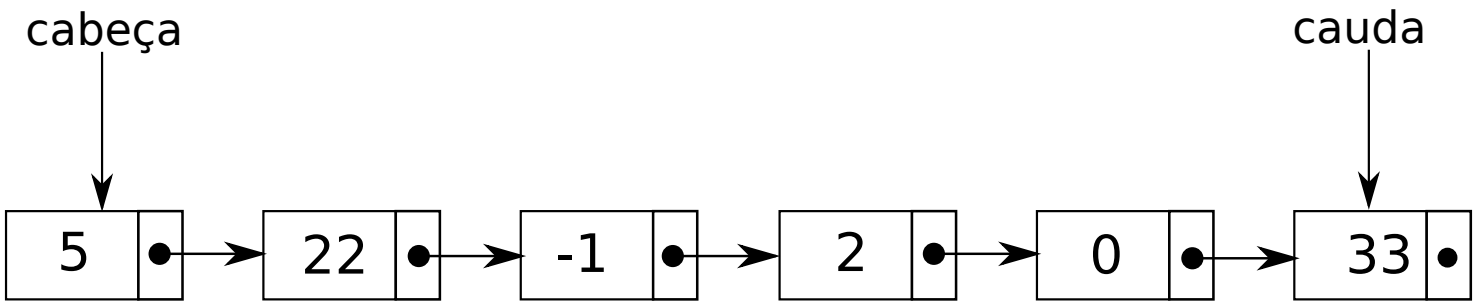


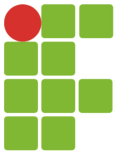


Listas Encadeadas: Inserção na Cauda

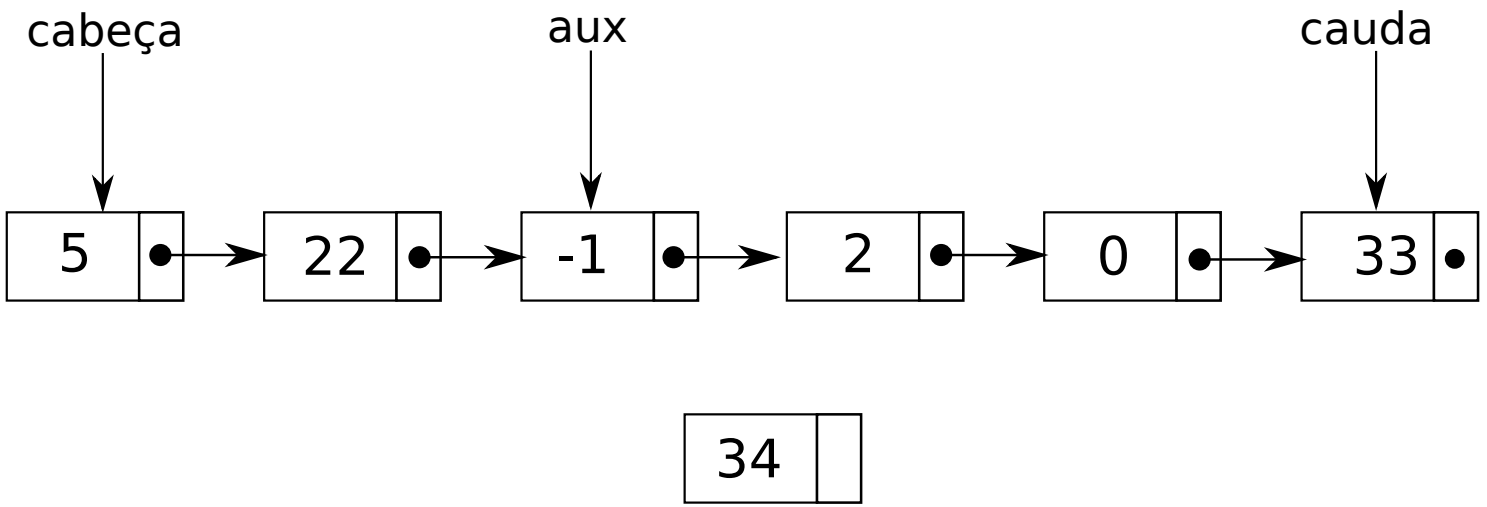
```
127 /** Insere um elemento na cauda da lista **/
128 void list_append(list_t* l, void* data){
129     /** Cria o novo nó ao chamar list_new_node **/
130     list_node_t* new_node = list_new_node(data,l->constructor);
131     /** Se a lista está vazia, a cabeça deve apontar para o nó recém criado **/
132     if(list_empty(l)){
133         l->head = new_node;
134     }
135     /** Caso contrário, a lista possui uma cauda e ela deve estabelecer
136      * uma ligação o elemento recém criado **/
137     else{
138         l->tail->next = new_node;
139     }
140     /** A cauda é atualizada para apontar para o elemento recém criado **/
141     l->tail = new_node;
142     /** O tamanho da lista é incrementado **/
143     l->size++;
144 }
```

Listas Encadeadas: Inserção

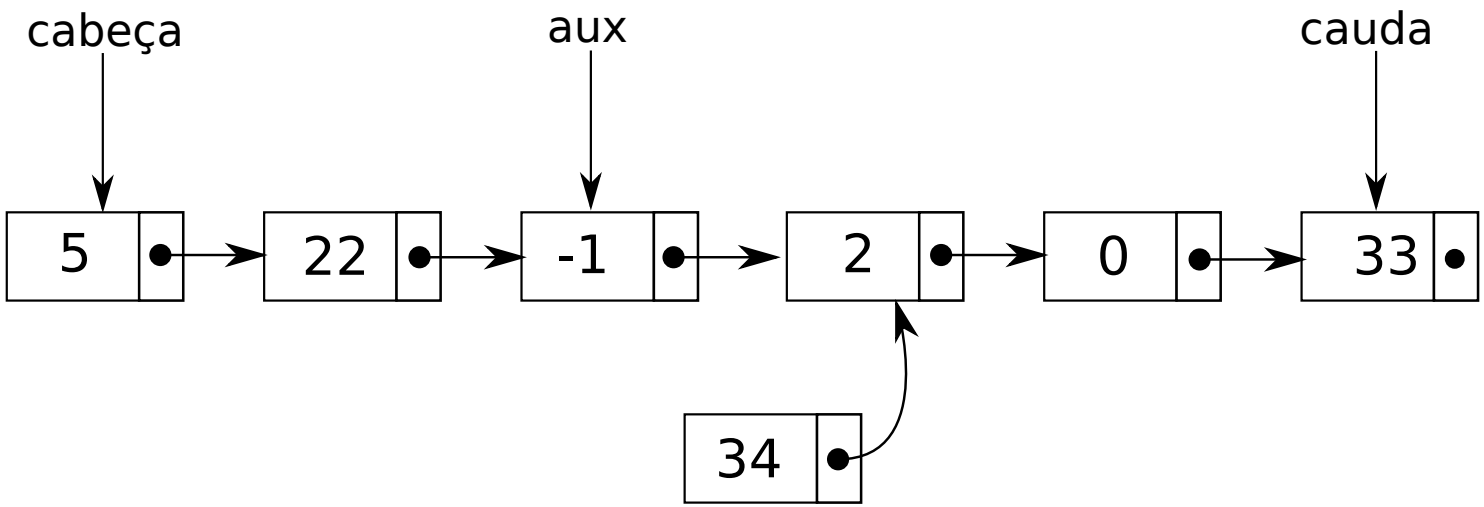


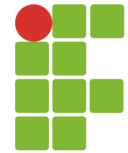


Listas Encadeadas: Inserção

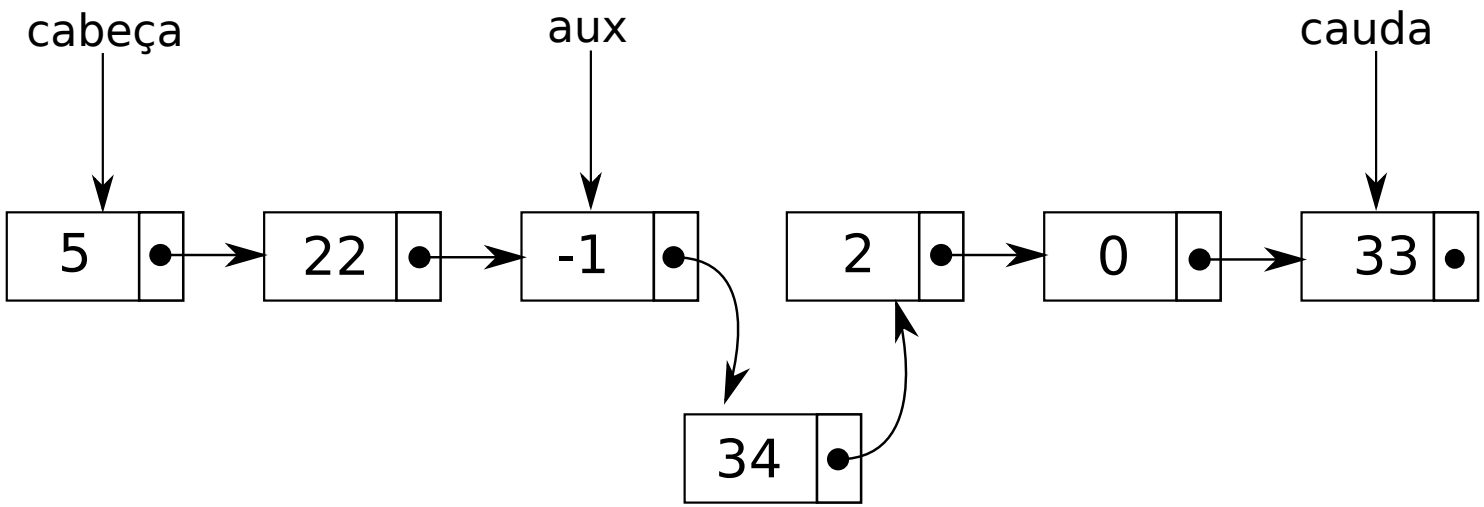


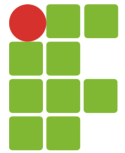
Listas Encadeadas: Inserção





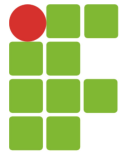
Listas Encadeadas: Inserção





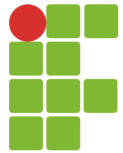
Listas Encadeadas: Inserção

```
74  /** Insere um novo dado na lista com base na posição i **/
75  void list_insert(list_t* l,void* data,size_t i){
76      /** Apenas modo debug, aborta o programa se a posição for inválida **/
77      assert(i<=list_size(l));
78      /** Se a lista está vazia, ou a posição de inserção é a 0, a
79                  inserção é feita na cabeça **/
80      if(list_empty(l) || i==0){
81          list_prepend(l, data);
82      }
83      /** Inserção na cauda **/
84      else if(i==list_size(l)){
85          list_append(l, data);
86      }
```



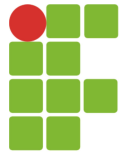
Listas Encadeadas: Inserção

```
87      /** Inserção no meio da lista que tem pelo menos 1 elemento **/
88      else{
89          /** Cria o novo nó ao chamar a função list_new_node **/
90          list_node_t* new_node = list_new_node(data,l->constructor);
91          /** Precisamos encontrar o elemento que antecede a posição i ao
92           * caminhar na lista **/
93          list_iterator_t it = l->head;
94          size_t k;
95          /** Caminhamos até a posição i-1 da lista **/
96          for(k=0;k<i-1;k++){
97              it = it->next;
98          }
99          /** it agora aponta pro elemento da posição i-1*/
100         /** Estabelecemos o next do novo nó para o elemento antigo da
101          * da posição i **/
102         new_node->next = it->next;
103         /** O next do nó da posição i-1 recebe o elemento recém inserido **/
104         it->next = new_node;
105         /** O tamanho da lista é incrementado **/
106         l->size++;
107     }
108 }
```



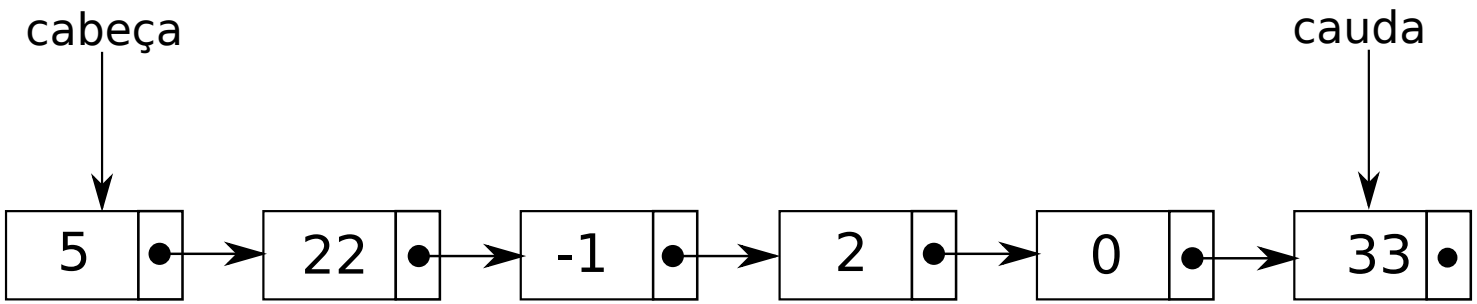
Listas Encadeadas: Inserção

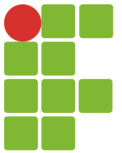
```
37  /** Deleta o nó de uma lista **/
38  static void list_delete_node(list_node_t* n,list_node_destructor_fn destructor){
39      /** Chama a função destrutora para o dado do nó **/
40      destructor(n->data);
41      /** Libera o nó **/
42      free(n);
43  }
```



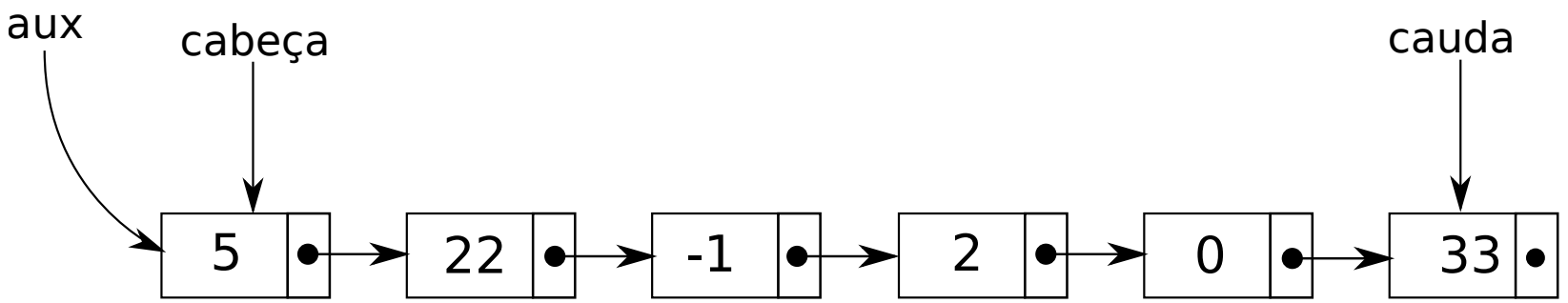
Sumário

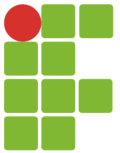
Listas Encadeadas: Remoção na Cabeça



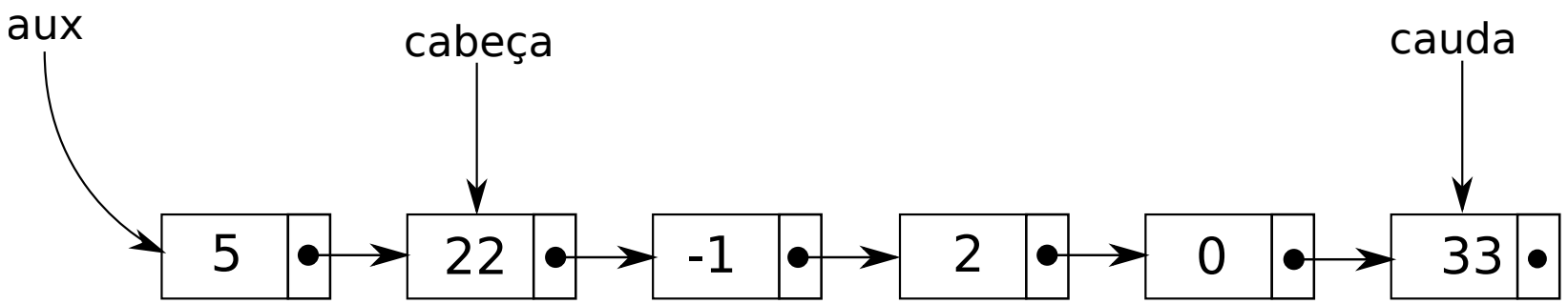


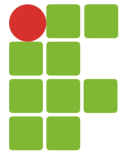
Listas Encadeadas: Remoção na Cabeça



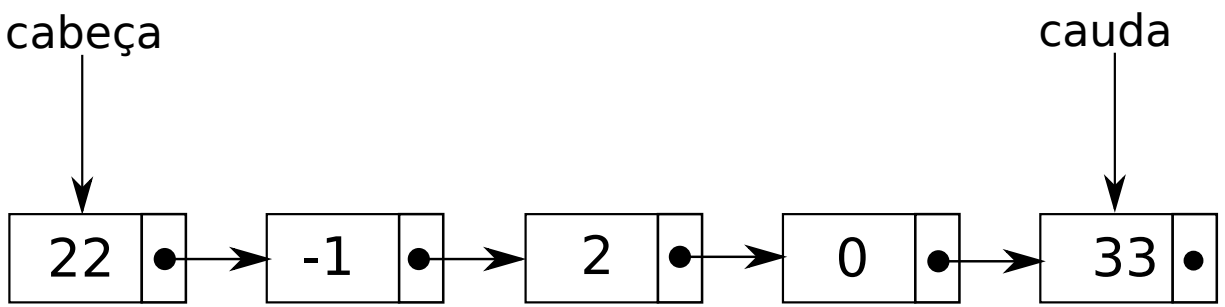


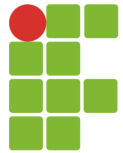
Listas Encadeadas: Remoção na Cabeça





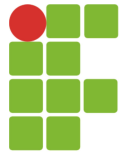
Listas Encadeadas: Remoção na Cabeça



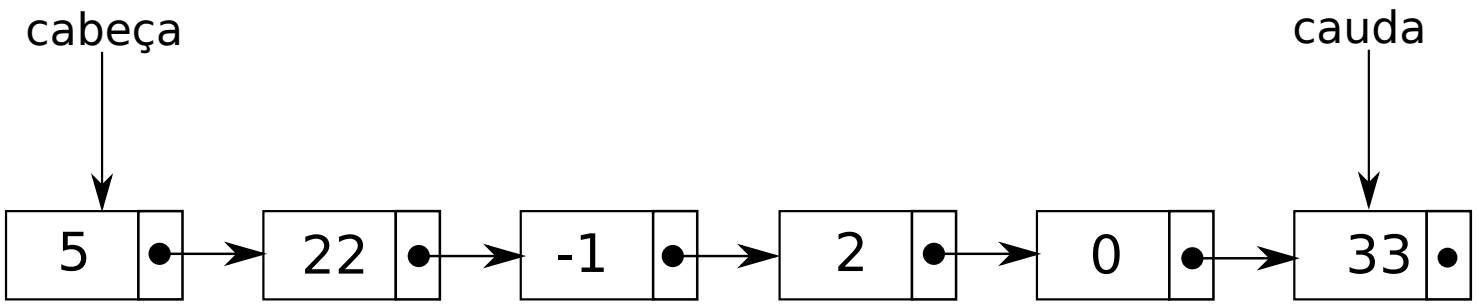


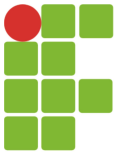
Listas Encadeadas: Remoção na Cabeça

```
185 void list_remove_head(list_t* l){
186     /** Debug apenas: aborta o programa caso a remoção da cabeça seja sobre
187      * uma lista vazia **/
188     assert(!list_empty(l));
189     /** O nó a ser removido recebe a cabeça **/
190     list_iterator_t node = l->head;
191     /** Se a lista tem um elemento, após a remoção a cauda deve ser NULL **/
192     if(list_size(l)==1){
193         l->tail = NULL;
194     }
195     /** A cabeça passa para o próximo elemento **/
196     l->head = l->head->next;
197     /** Deleta-se a cabeça **/
198     list_delete_node(node,l->destructor);
199     /** O tamanho da lista é decrementado **/
200     l->size--;
201 }
```

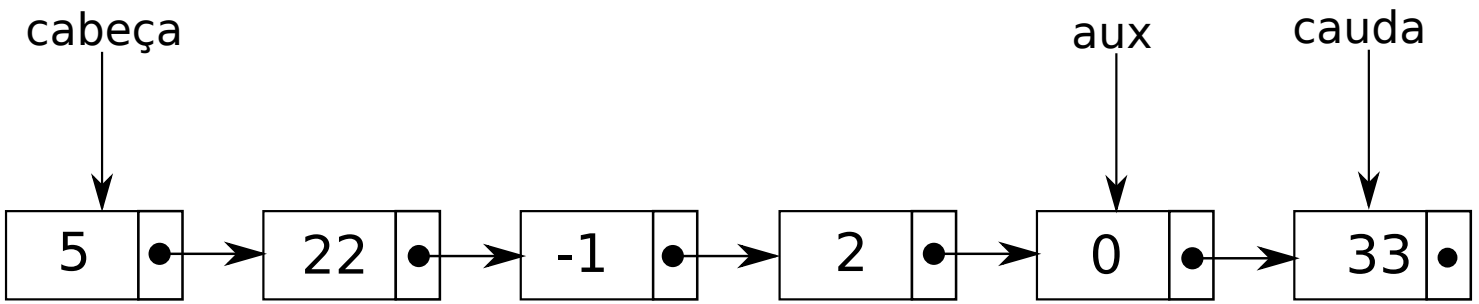


Listas Encadeadas: Remoção na Cauda

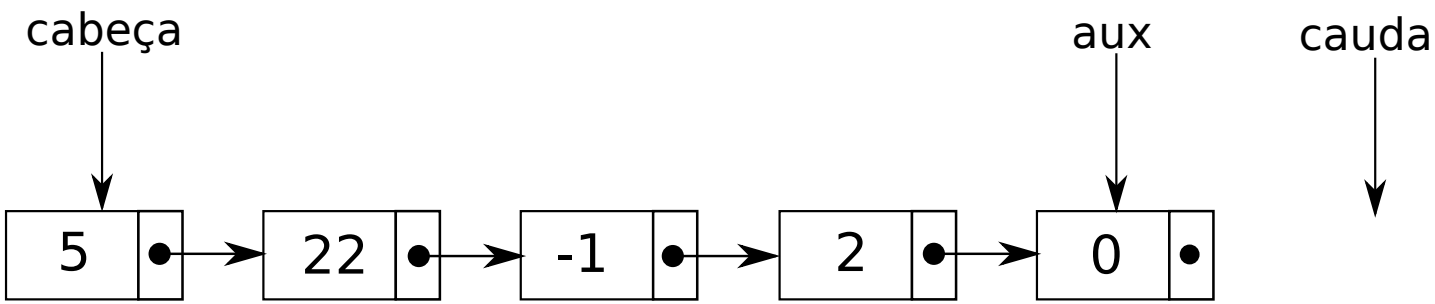




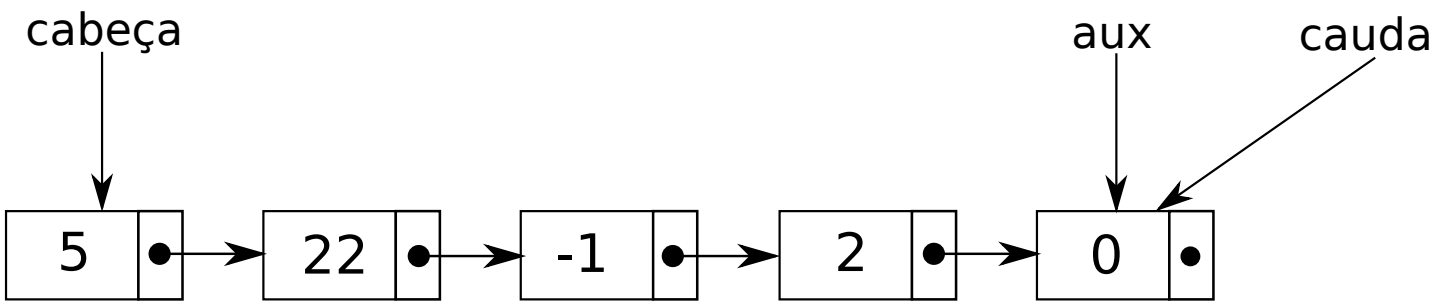
Listas Encadeadas: Remoção na Cauda

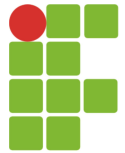


Listas Encadeadas: Remoção na Cauda



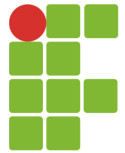
Listas Encadeadas: Remoção na Cauda





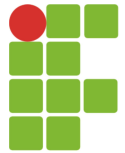
Listas Encadeadas: Remoção na Cauda

```
203 /** Remove a cauda da lista **/
204 void list_remove_tail(list_t* l){
205     /** Debug apenas,aborta o programa caso a função seja chamada para uma
206      * lista vazia **/
207     assert(list_size(l)>0);
208     /** O nó a ser removido recebe a cauda **/
209     list_iterator_t node = l->tail;
210     /** Se a lista tem tamanho 1, a cauda e a cabeça apontam para NULL
211      * após a remoção **/
212     if(list_size(l)==1){
213         l->head = NULL;
214         l->tail = NULL;
215     }
216     /** Caso contrário, a lista tem mais de um elemento. Deve-se iterar sobre
217      * a lista até o penúltimo elemento **/
```

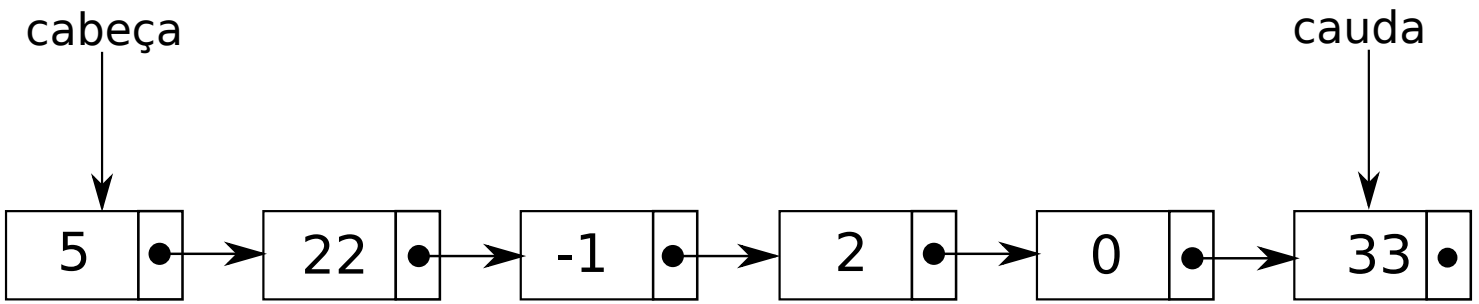


Listas Encadeadas: Remoção na Cauda

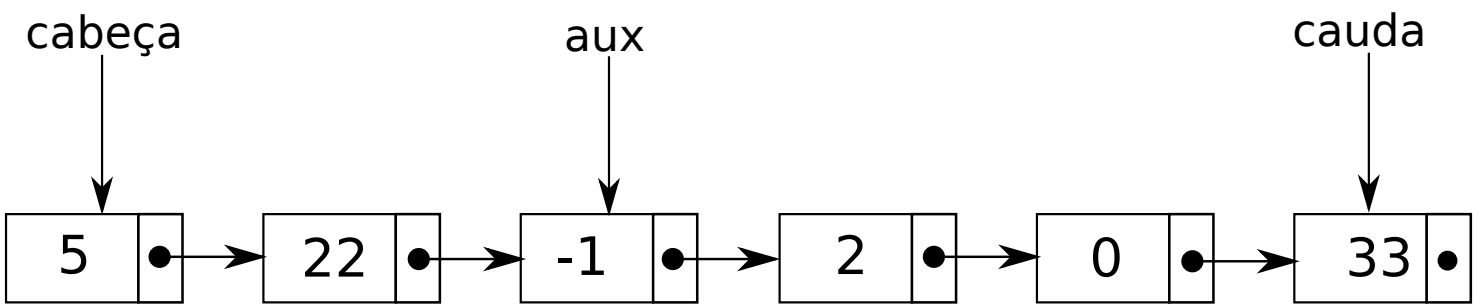
```
218     else{
219         /** Itera-se sobre a lista a partir da cabeça até o penúltimo elemento
220          **/
221         list_iterator_t it = l->head;
222         while(it->next!=l->tail){
223             it = it->next;
224         }
225         /** O campo next do penúltimo elemento agora aponta para NULL **/
226         it->next = NULL;
227         /** O penúltimo elemento passa a ser a cauda **/
228         l->tail = it;
229     }
230     /** Remove-se a cauda antiga **/
231     list_delete_node(node,l->destructor);
232     /** O tamanho da lista é decrementado **/
233     l->size--;
234 }
235
236 /** Acessa o i-ésimo elemento da lista **/
```



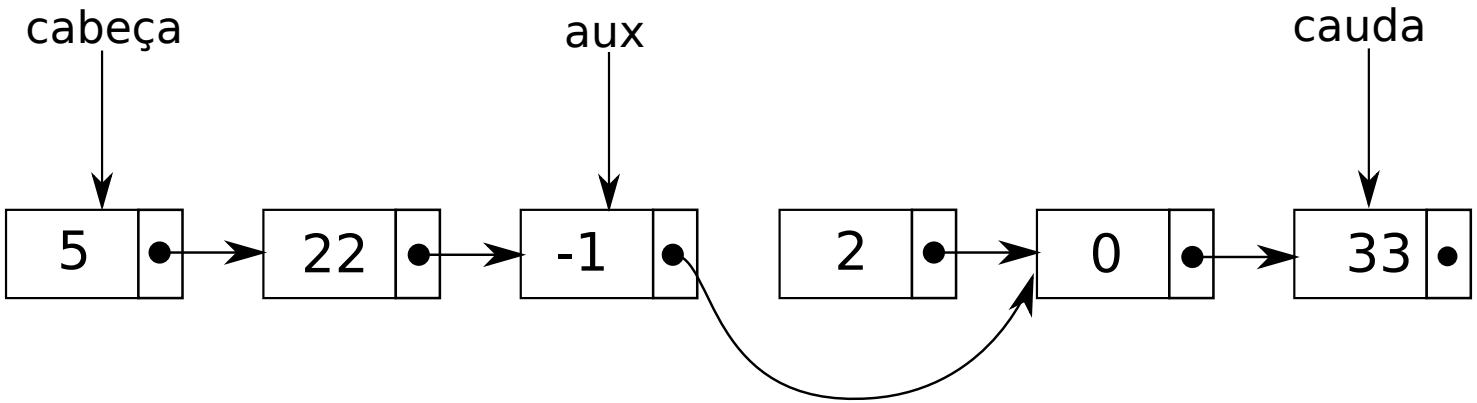
Listas Encadeadas: Remoção



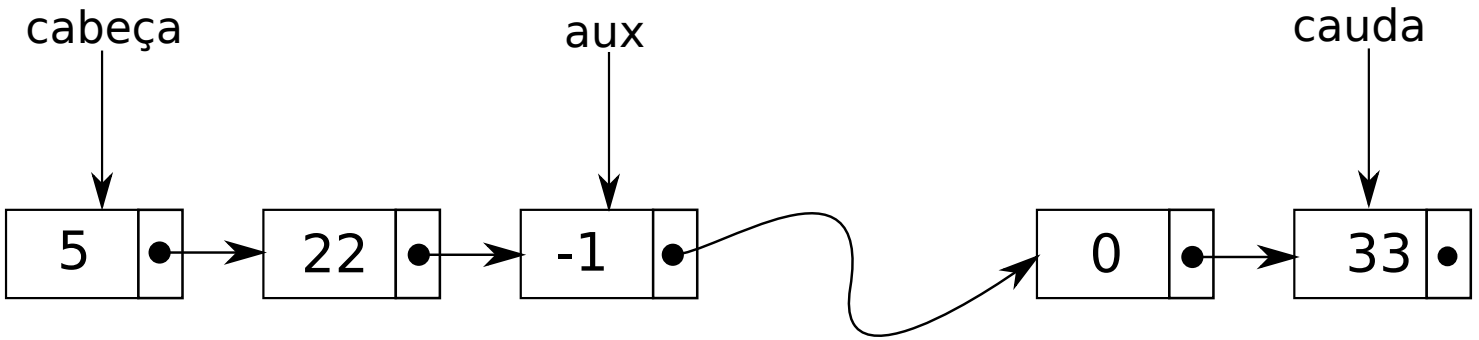
Listas Encadeadas: Remoção

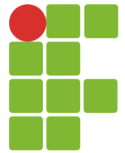


Listas Encadeadas: Remoção



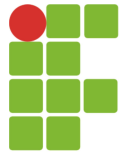
Listas Encadeadas: Remoção





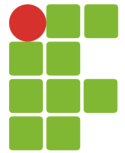
Listas Encadeadas: Remoção

```
146 /** Remove o elemento da posição i da lista **/
147 void list_remove(list_t* l,size_t i){
148     /** Debug apenas, aborta o programa se a remoção estiver sendo feita
149      * em uma lista vazia ou em uma posição inexistente da lista **/
150     assert(!list_empty(l) && i<list_size(l));
151     /** Se a lista tem tamanho 1, ou a remoção é do primeiro elemento,
152                 equivale a eliminar a cabeça
153         **/
154     if(list_size(l)==1 || i==0){
155                 list_remove_head(l);
156     }
157     /** Se i==list_size(l)-1, a remoção é na cauda **/
158     else if(i==list_size(l)-1){
159         list_remove_tail(l);
160     }
```

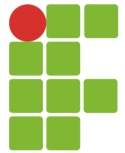


Listas Encadeadas: Remoção

```
161     /** O nó a ser removido encontra-se no meio da lista e a lista possui mais que um elemento **/
162     else{
163         /** Nó a ser removido **/
164         list_node_t* node;
165         /** Devemos percorrer até o i-1-ésimo elemento a partir da cabeça **/
166         list_iterator_t it = l->head;
167         size_t k;
168         /** Itera-se até o elemento imediatamente interior ao elemento i **/
169         for(k=0;k<i-1;k++){
170             it = it->next;
171         }
172         /** Nó a ser removido passa a ser o i-ésimo elemento **/
173         node = it->next;
174         /** O anterior ao nó a ser removido aponta para o elemento
175          * que vem após o nó a ser removido **/
176         it->next = node->next;
177         /** Deleta o nó atribuido anteriormente **/
178         list_delete_node(node,l->destructor);
179         /** Decrementa o tamanho da lista **/
180         l->size--;
181     }
182 }
```

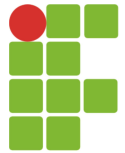


Sumário



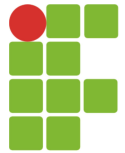
Listas Encadeadas: Acesso

- Acesso na cabeça ou na cauda é fácil. Já temos ponteiros para estas posições.
- Para acessar uma posição arbitrária, começamos da cabeça e iteramos na lista até posicionarmos o ponteiro na posição em que queremos acessar.
- Diferentemente de vetores, listas encadeadas não possuem acesso direto (aleatório).



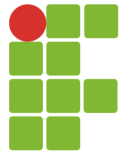
Listas Encadeadas: Acesso

```
259 /** Acessa a cabeça da lista **/
260 void* list_access_head(list_t* l){
261     /** Debug apenas, aborta o programa se a lista não tem cabeça (é vazia) **/
262     assert(!(list_empty(l)));
263     return (l->head->data);
264 }
```



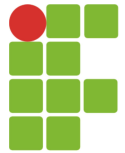
Listas Encadeadas: Acesso

```
266 /** Acessa a cauda da lista **/
267 void* list_access_tail(list_t* l){
268     /** Debug apenas, aborta o programa se a lista não tem cauda (é vazia)**/
269     assert(!list_empty(l));
270     return (l->tail->data);
271 }
```



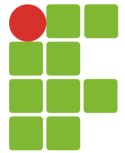
Listas Encadeadas: Acesso

```
236 /** Acessa o i-ésimo elemento da lista **/
237 void* list_access(list_t* l,size_t i){
238     /** Debug apenas, aborta o programa em caso de posição inválida
239      * a ser acessada **/
240     assert(!list_empty(l) && i<list_size(l));
241     /** Se i==0, o acesso é na cabeça **/
242     if(i==0){
243         return(list_access_head(l));
244     }
245     /** Se i==list_size(l)-1, o acesso é na cauda **/
246     else if(i==list_size(l)-1){
247         return(list_access_tail(l));
248     }
249     /** Caso contrário, percorre-se a lista até o i-ésimo elemento **/
250     size_t j;
251     list_iterator_t it = l->head;
252     for(j=0;j<i;j++){
253         it = it->next;
254     }
255     /** O campo dado do elemento acessado é retornado **/
256     return(it->data);
257 }
```

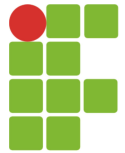


Listas Encadeadas: Acesso

```
62  /** Deleta uma lista com sucessivas remoções da cabeça **/
63  void list_delete(list_t** l){
64      /** Enquanto a lista não for fazia, remove a cabeça **/
65      while(!list_empty(*l)){
66          list_remove_head(*l);
67      }
68      /** Desaloca espaço para a estrutura de dados **/
69      free(*l);
70      /** Atribui NULL ao ponteiro para nossa lista **/
71      *l = NULL;
72  }
```

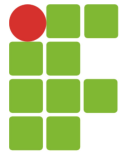


Sumário



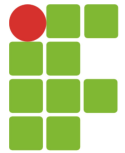
Listas Encadeadas: Limpeza

- Para deletar a lista da memória, basta iterar sobre ela e apagar os nós.
- Só devemos ter cuidado de não perder a referência para o próximo nó.
- Uma estratégia é sempre apagar a cabeça da lista enquanto ela não é vazia.

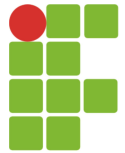


Listas Encadeadas: Limpeza

```
63  void list_delete(list_t** l){
64      /** Enquanto a lista não for fazia, remove a cabeça **/
65      while(!list_empty(*l)){
66          list_remove_head(*l);
67      }
68      /** Desaloca espaço para a estrutura de dados **/
69      free(*l);
70      /** Atribui NULL ao ponteiro para nossa lista **/
71      *l = NULL;
72  }
```

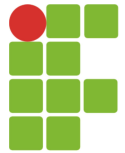


Sumário

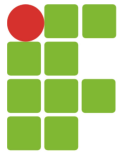


Listas Encadeadas

Operação	Complexidade
Inserção na cabeça	$\Theta(1)$
Inserção na cauda	$\Theta(1)$
Inserção em posição arbitrária	$\Theta(n)$
Remoção da cabeça	$\Theta(1)$
Remoção da cauda	$\Theta(n)$
Remoção de uma posição arbitrária	$\Theta(n)$
Acesso à cabeça	$\Theta(1)$
Acesso à cauda	$\Theta(1)$
Acesso à posição arbitrária	$\Theta(n)$

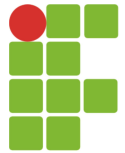


Sumário



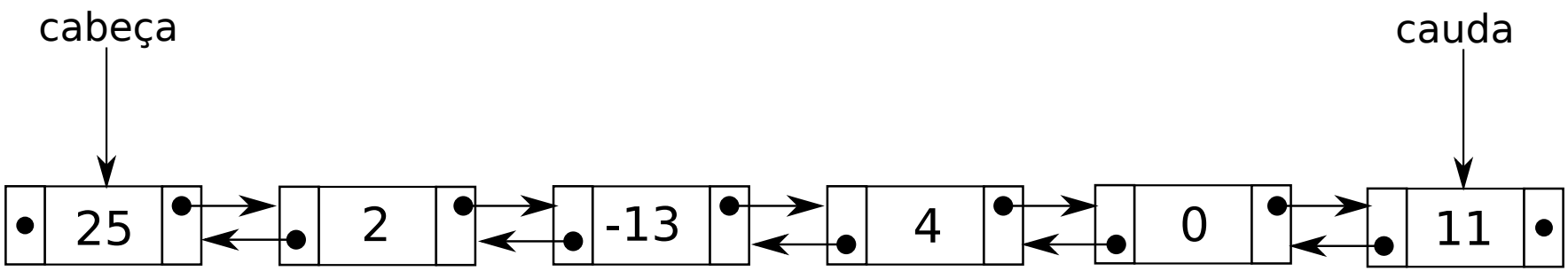
Listas Duplamente Encadeadas

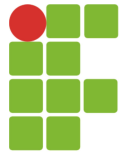
- Listas duplamente encadeadas se assemelham muito às listas encadeadas com a diferença que cada elemento possui uma referência para o elemento anterior.
- Apesar de utilizar mais espaço para representação, pode-se caminhar no sentido contrário.
- As operações em Listas Duplamente encadeada são **similares** às das Listas Encadeadas, com atenção para atualizar o ponteiro do elemento anterior.



Sumário

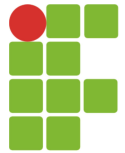
Listas Duplamente Encadeadas





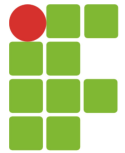
Listas Duplamente Encadeadas: Definição

```
6  typedef void* (*dlist_node_constructor_fn) (void*);
7  typedef void (*dlist_node_destructor_fn)(void *);
```



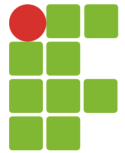
Listas Duplamente Encadeadas: Definição

```
10  /**A nossa dlista encadeada consiste de vários nós,
11  que possuem o tipo linked_dlist_node_t **/
12  typedef struct dlist_node_t{
13      void* data; /*Ponteiro para um dado genérico de qualquer tipo*/
14      struct dlist_node_t* next; /*ponteiro para o próximo elemento*/
15      struct dlist_node_t* prev; /*Ponteiro para o elemento anterior*/
16  }dlist_node_t;
```



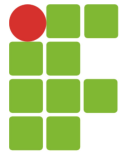
Listas Duplamente Encadeadas: Definição

```
21 typedef struct dlist_t{
22     dlist_node_t* head; /*Cabeça da dlista*/
23     dlist_node_t* tail; /*Cauda da dlista*/
24     dlist_node_constructor_fn constructor; /*Função para construir o objeto*/
25     dlist_node_destructor_fn destructor; /*Função para destruir o objeto*/
26     size_t size; /*tamanho da dlista*/
27 }dlist_t;
```

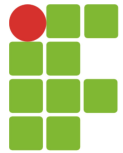


Listas Duplamente Encadeadas: Definição

```
19 typedef dlist_node_t* dlist_iterator_t;
```

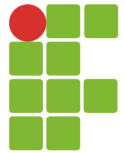


Sumário

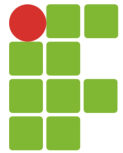


Listas Duplamente Encadeadas: Inicialização

```
23  /**Inicializa a lista duplamente encadeada e seus membros**/
24  void dlist_initialize(dlist_t** l,dlist_node_constructor_fn constructor,
25      dlist_node_destructor_fn destructor){
26      (*l) = mallocx(sizeof(dlist_t));
27      (*l)->head = NULL;
28      (*l)->tail = NULL;
29      (*l)->size = 0;
30      (*l)->constructor = constructor;
31      (*l)->destructor = destructor;
32  }
```

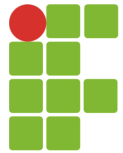


Sumário



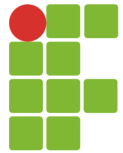
Listas Duplamente Encadeadas: Funções Auxiliares

```
170 /**Retorna o tamanho da dlista**/
171 size_t dlist_size(dlist_t* l){
172     return l->size;
173 }
```



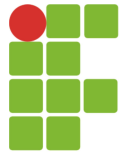
Listas Duplamente Encadeadas: Funções Auxiliares

```
175 /**Retorna verdadeiro se a dlista está vazia, e falso caso contrário**/
176 size_t dlist_empty(dlist_t* l){
177     return dlist_size(l)==0 ? 1 : 0;
178 }
```



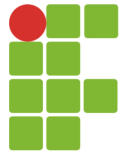
Listas Duplamente Encadeadas: Funções Auxiliares

```
8  static dlist_node_t* dlist_new_node(void* data,dlist_node_constructor_fn constructor){
9      dlist_node_t* new_node =  mallocx(sizeof(dlist_node_t));
10     new_node->data = constructor(data);
11     new_node->next = NULL;
12     new_node->prev = NULL;
13     return new_node;
14 }
```

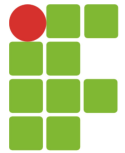


Listas Duplamente Encadeadas: Funções Auxiliares

```
17  static void dlist_delete_node(dlist_node_t* node,dlist_node_destructor_fn destructor){
18      destructor(node->data);
19      free(node);
20  }
```



Sumário

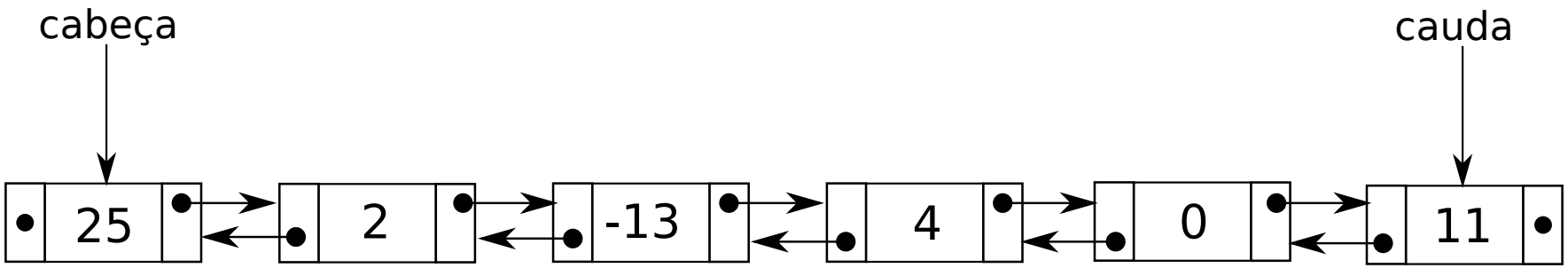


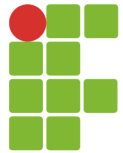
Listas Duplamente Encadeadas: Inserção

Inserção na Cabeça e Cauda

- Igual às versões das listas encadeadas.
- Só precisamos de cuidado para atualizar os ponteiros que ligam ao próximo ou ao anterior.

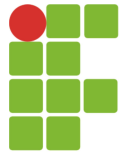
Listas Duplamente Encadeadas: Inserção na Cabeça e na Cauda





Listas Duplamente Encadeadas: Inserção na Cabeça e na Cauda

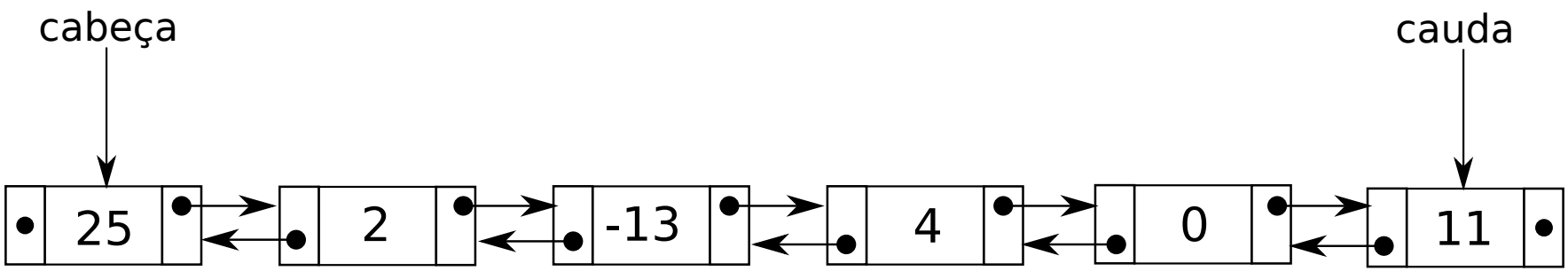
```
69  /** Insere um elemento na cabeça da dlista **/
70  void dlist_prepend(dlist_t* l,void* data){
71      dlist_node_t* new_node = dlist_new_node(data,l->constructor);
72      if(dlist_empty(l)){
73          l->tail = new_node;
74      }
75      else{
76          l->head->prev = new_node;
77      }
78      new_node->next = l->head;
79      l->head = new_node;
80      l->size++;
81  }
```



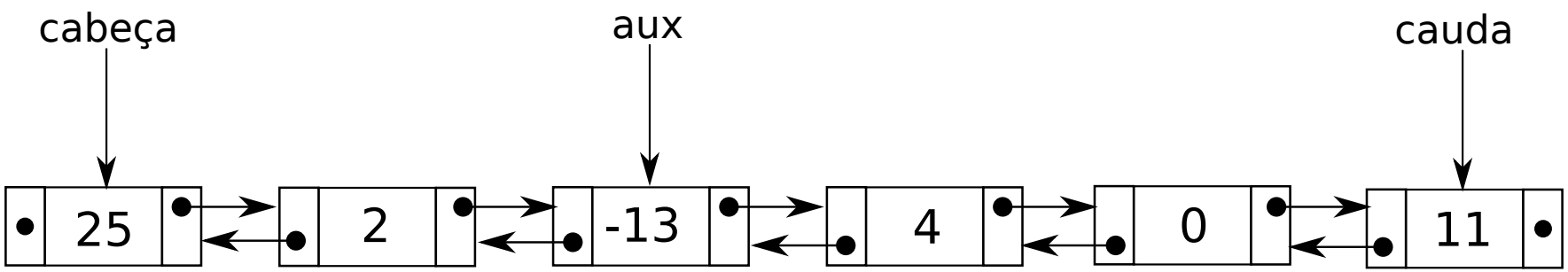
Listas Duplamente Encadeadas: Inserção na Cabeça e na Cauda

```
83  /**Insere um elemento na cauda da dlista **/
84  void dlist_append(dlist_t* l, void* data){
85      dlist_node_t* new_node = dlist_new_node(data,l->constructor);
86      if(dlist_empty(l)){
87          l->head = new_node;
88      }
89      else{
90          l->tail->next = new_node;
91      }
92      new_node->prev = l->tail;
93      l->tail = new_node;
94      l->size++;
95  }
```

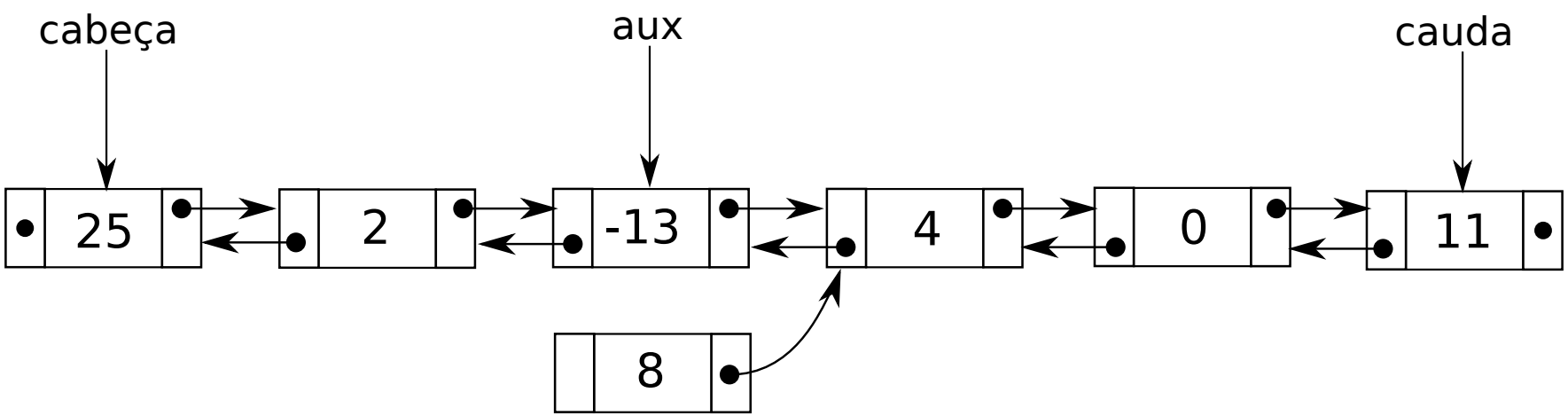
Listas Duplamente Encadeadas: Inserção



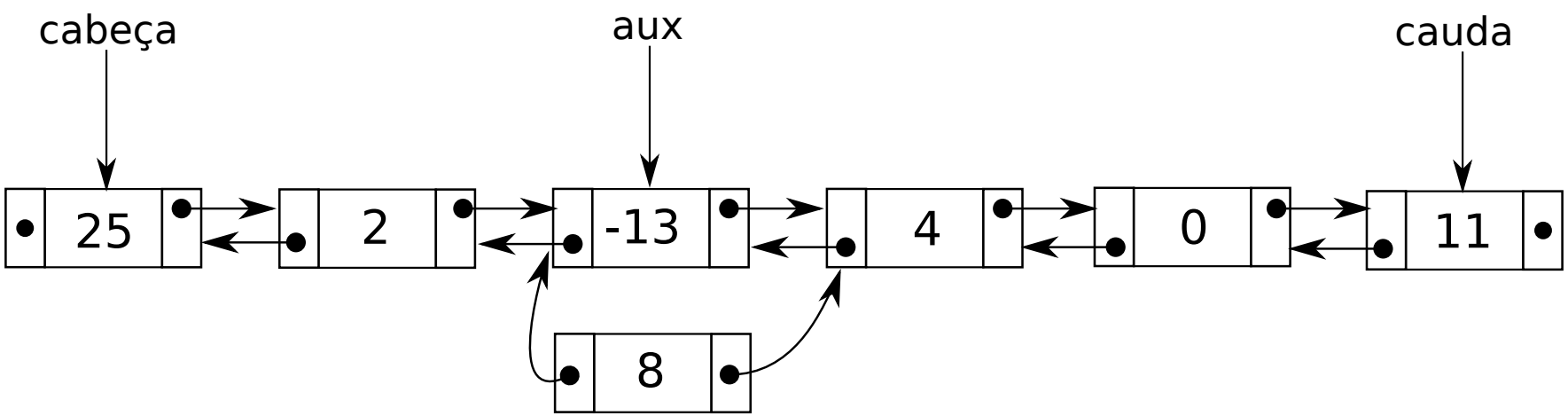
Listas Duplamente Encadeadas: Inserção



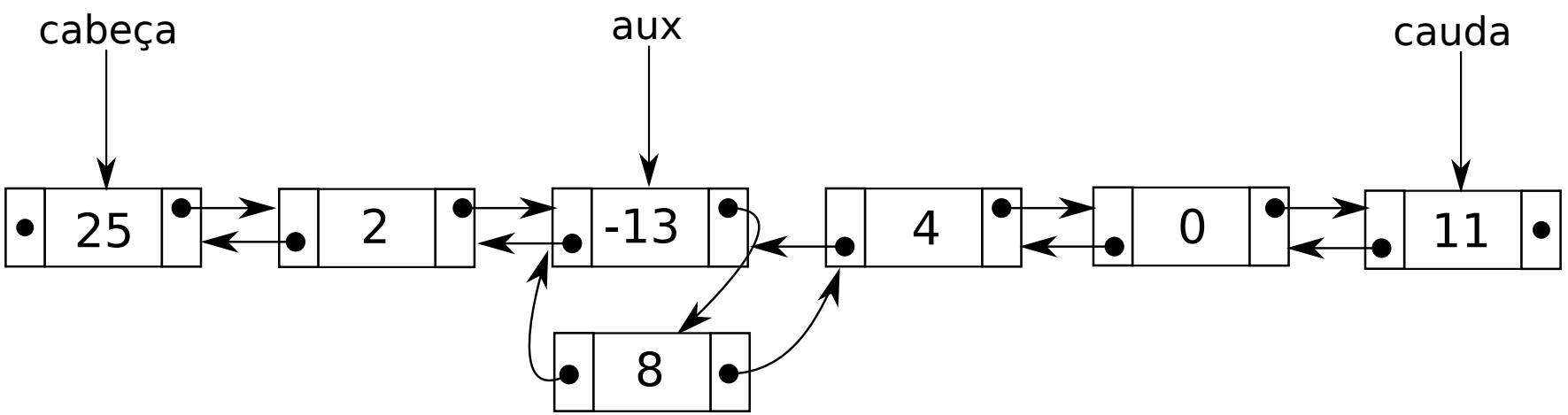
Listas Duplamente Encadeadas: Inserção



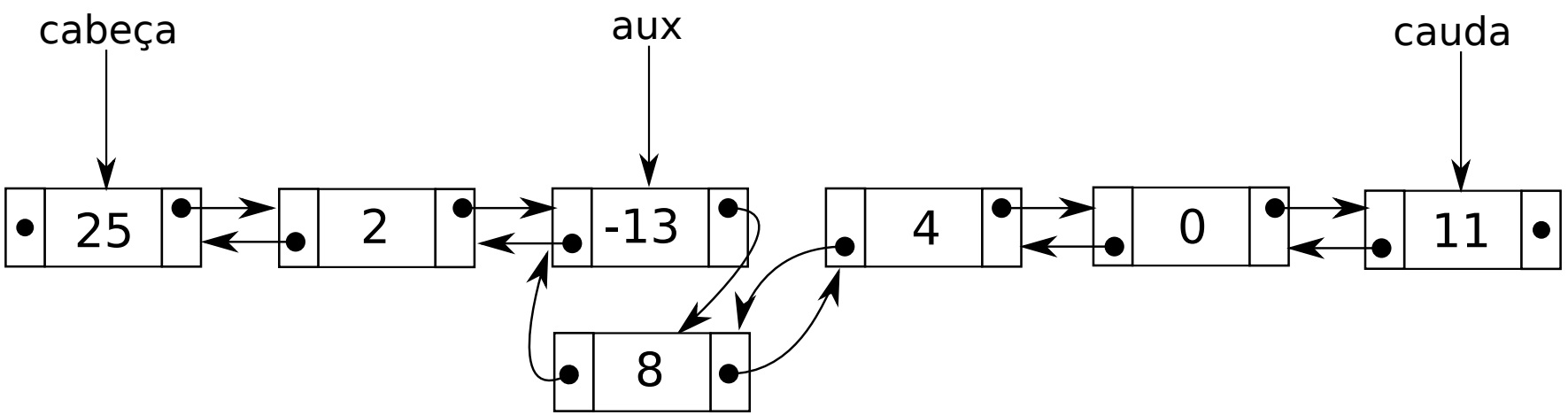
Listas Duplamente Encadeadas: Inserção

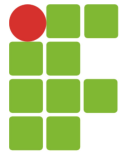


Listas Duplamente Encadeadas: Inserção



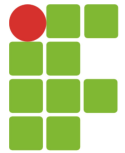
Listas Duplamente Encadeadas: Inserção





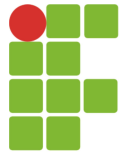
Listas Duplamente Encadeadas: Inserção

```
42  void dlist_insert(dlist_t* l,void* data,size_t i){
43      assert(i<=dlist_size(l));
44      if(dlist_empty(l) || i==0){
45          dlist_prepend(l,data);
46      }
47      else if(i==dlist_size(l)){
48          /*Inserção na cauda*/
49          dlist_append(l,data);
50      }
```

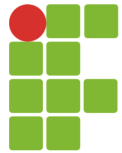


Listas Duplamente Encadeadas: Inserção

```
51      else{
52          dlist_node_t* new_node = dlist_new_node(data,l->constructor);
53          /*Inserção no meio da lista*/
54          /*Precisamos encontrar o elemento que antecede a posição i*/
55          dlist_iterator_t it = l->head;
56          size_t k;
57          for(k=0;k<i-1;k++){
58              it = it->next;
59          }
60          /*aux agora aponta pro elemento da posição i-1*/
61          new_node->next = it->next;
62          new_node->prev = it;
63          it->next->prev = new_node;
64          it->next = new_node;
65          l->size++;
66      }
67  }
```



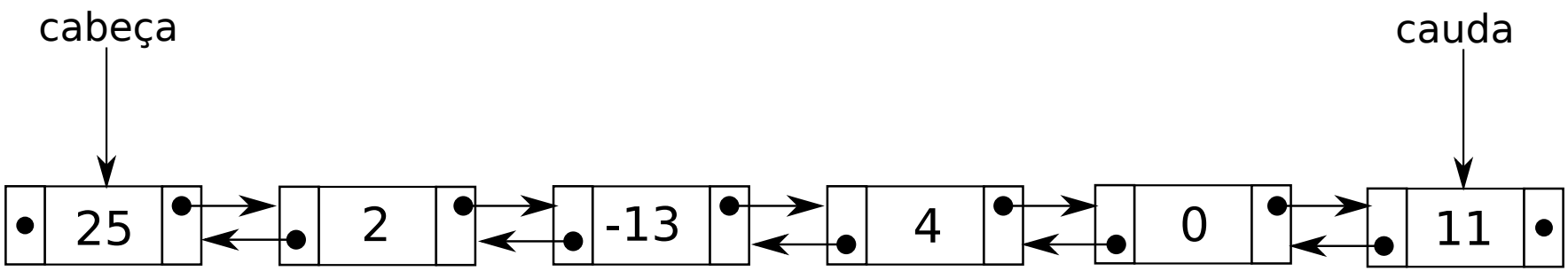
Sumário

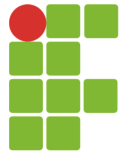


Listas Duplamente Encadeadas: Remoção na Cabeça

- Igual a versão das listas encadeadas.
- Só precisamos de cuidado para atualizar os ponteiros adicionais.

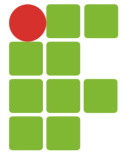
Listas Duplamente Encadeadas: Remoção na Cabeça





Listas Duplamente Encadeadas: Remoção na Cabeça

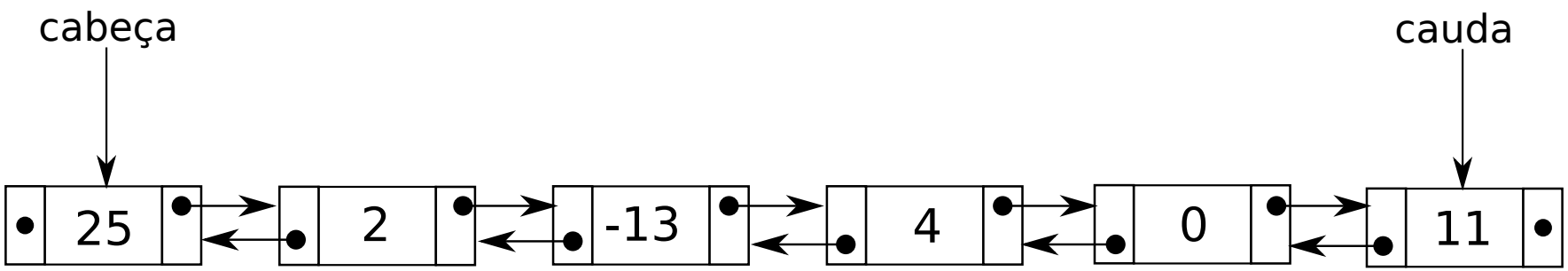
```
122 void dlist_remove_head(dlist_t* l){
123     assert(!dlist_empty(l));
124     dlist_iterator_t node = l->head;
125     l->head = l->head->next;
126     if(dlist_size(l)==1){
127         l->tail = NULL;
128     }
129     else{
130         l->head->prev = NULL;
131     }
132     dlist_delete_node(node,l->destructor);
133     l->size--;
134 }
```

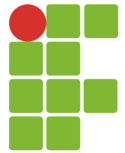


Listas Duplamente Encadeadas: Remoção na Cauda

- Na versão da lista encadeada simples, precisávamos percorrer a lista toda até o penúltimo elemento.
- Como em listas duplamente encadeadas conseguimos acessar o penúltimo elemento ao começar do último e utilizar o ponteiro para o anterior, o penúltimo elemento é obtido em tempo constante!
- $\Theta(n) \Rightarrow \Theta(1)$.
- O restante da remoção é igual ao da lista encadeada simples, com cuidado de atualizar os ponteiros adicionais.

Listas Duplamente Encadeadas: Remoção na Cauda

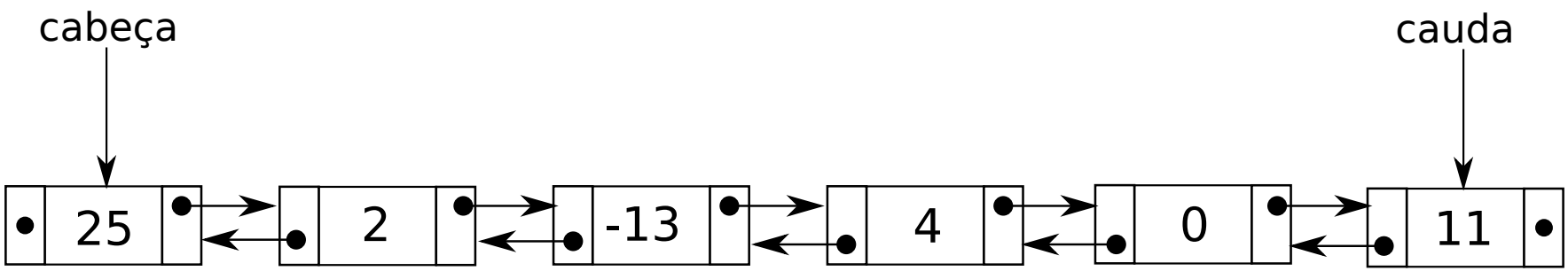




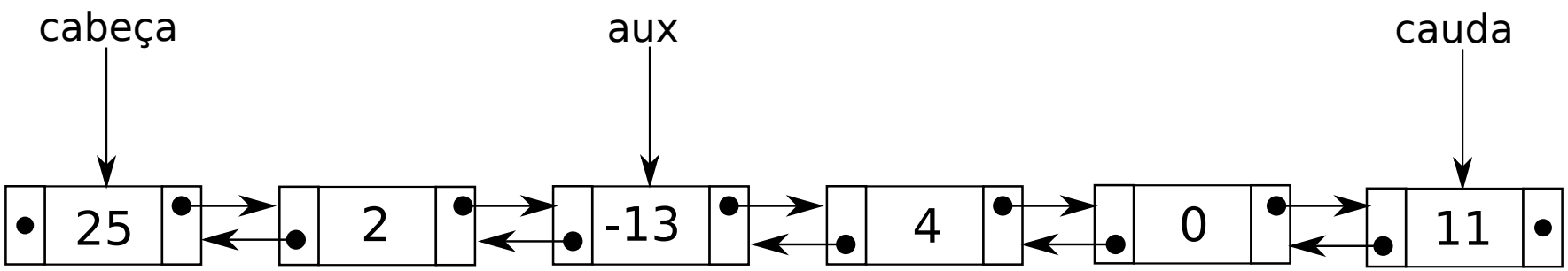
Listas Duplamente Encadeadas

```
136 void dlist_remove_tail(dlist_t* l){
137     assert(!dlist_empty(l));
138     dlist_iterator_t node = l->tail;
139     l->tail = l->tail->prev;
140     if(dlist_size(l)==1){
141         l->head = NULL;
142     }
143     else{
144         l->tail->next = NULL;
145     }
146     dlist_delete_node(node,l->destructor);
147     l->size--;
148 }
```

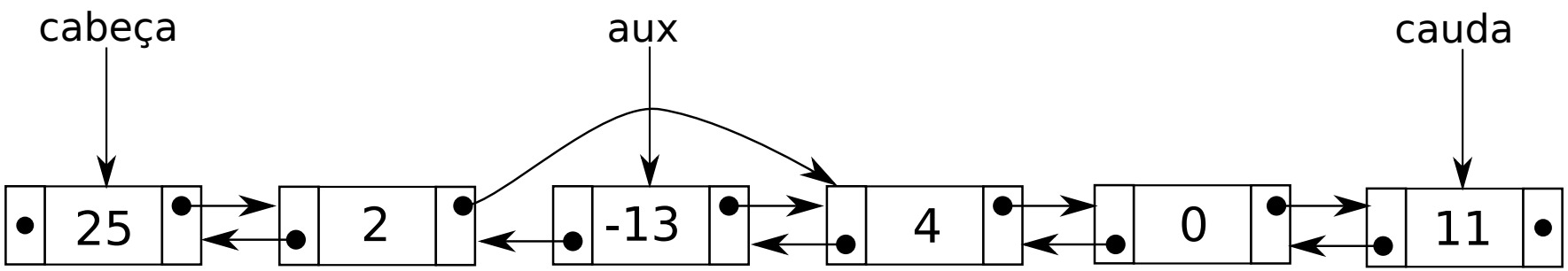
Listas Duplamente Encadeadas: Remoção



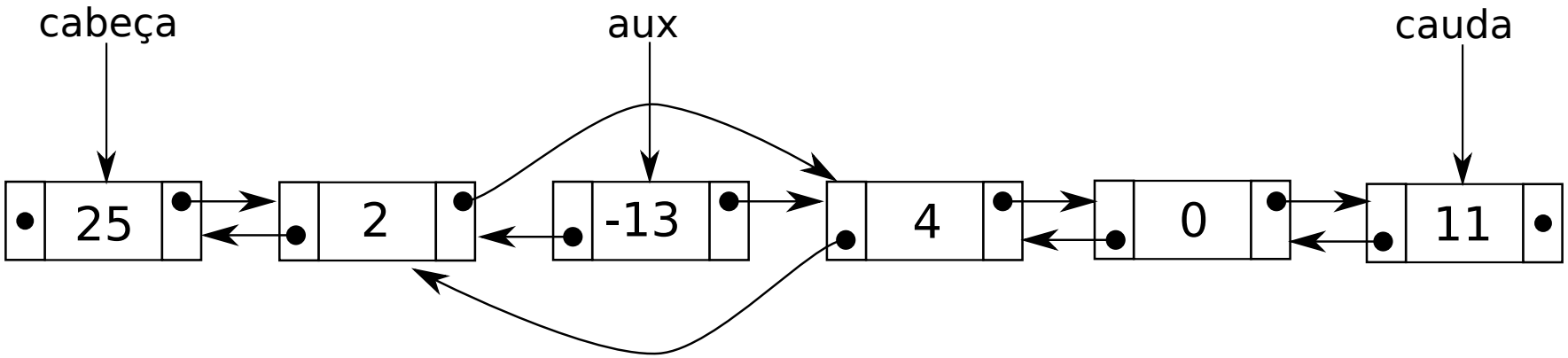
Listas Duplamente Encadeadas: Remoção



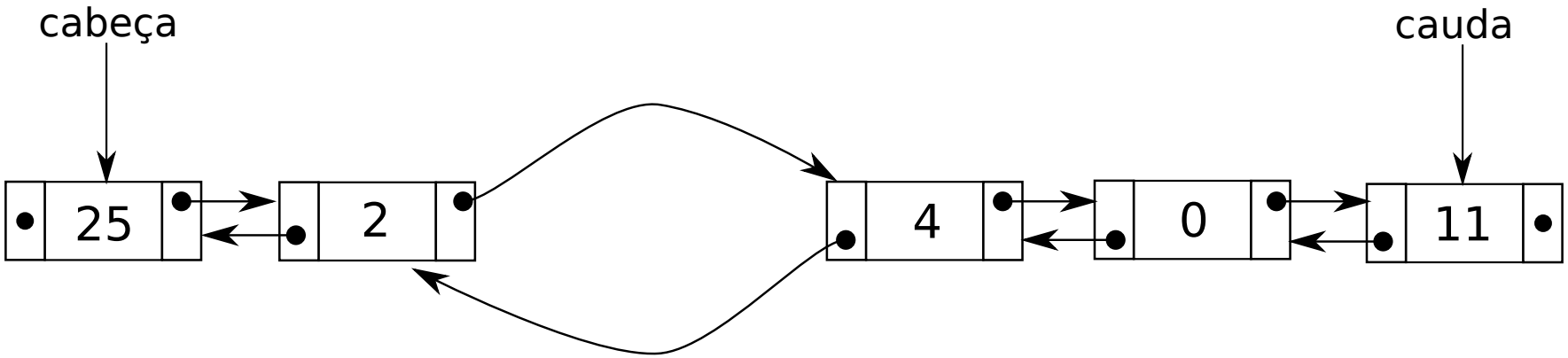
Listas Duplamente Encadeadas: Remoção

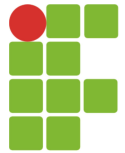


Listas Duplamente Encadeadas: Remoção



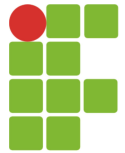
Listas Duplamente Encadeadas: Remoção





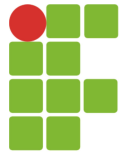
Listas Duplamente Encadeadas: Remoção

```
97  /**Remove o elemento da posição i da dlista**/
98  void dlist_remove(dlist_t* l,size_t i){
99      assert(!dlist_empty(l) && i<dlist_size(l));
100     dlist_node_t* node;
101     if(dlist_size(l)==1 || i==0){
102                 dlist_remove_head(l);
103         }
104     else if(i==dlist_size(l)-1){
105         dlist_remove_tail(l);
106     }
```

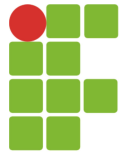


Listas Duplamente Encadeadas: Remoção

```
107     else{
108         dlist_iterator_t it = l->head;
109         size_t k;
110         for(k=0;k<i;k++){
111             it = it->next;
112         }
113         node = it;
114         node->prev->next = node->next;
115         node->next->prev = node->prev;
116         dlist_delete_node(node,l->destructor);
117         l->size--;
118     }
119
120 }
```

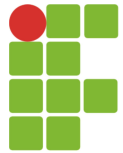


Sumário



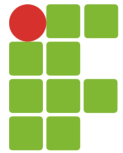
Listas Duplamente Encadeadas: Acesso

- O Acesso em listas duplamente encadeadas é análogo ao das listas encadeadas simples.
- Durante o acesso à uma posição arbitrária, podemos começar a busca pela cabeça ou pela cauda, a escolha dependerá de qual estará mais próxima da posição em que se deseja acesso.



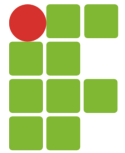
Listas Duplamente Encadeadas: Acesso

```
160 void* dlist_access_head(dlist_t* l){
161     assert(!(dlist_empty(l)));
162     return (l->head->data);
163 }
```



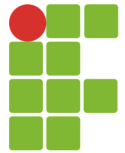
Listas Duplamente Encadeadas: Acesso

```
165 void* dlist_access_tail(dlist_t* l){
166     assert(!dlist_empty(l));
167     return (l->tail->data);
168 }
```

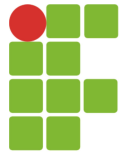


Listas Duplamente Encadeadas: Acesso

```
150 void* dlist_access(dlist_t* l,size_t i){
151     assert(!dlist_empty(l) && i<dlist_size(l));
152     dlist_iterator_t it = l->head;
153     size_t j;
154     for(j=0;j<i;j++){
155         it = it->next;
156     }
157     return(it->data);
158 }
```

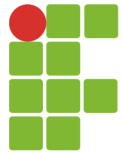


Sumário



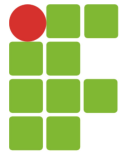
Listas Duplamente Encadeadas: Limpeza

- Funciona de forma análoga ao das listas encadeadas simples.

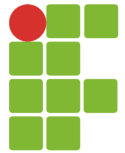


Listas Duplamente Encadeadas: Limpeza

```
33  /** Deleta a lista por inteiro e libera espaço em memória **/
34  void dlist_delete(dlist_t** l){
35      while(!dlist_empty(*l)){
36          dlist_remove_head(*l);
37      }
38      free(*l);
39      *l = NULL;
40  }
```



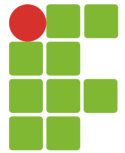
Sumário



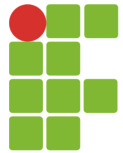
Listas Duplamente Encadeadas

Complexidade das Operações

Operação	Complexidade
Inserção na cabeça	$\Theta(1)$
Inserção na cauda	$\Theta(1)$
Inserção em posição arbitrária	$\Theta(n)$
Remoção da cabeça	$\Theta(1)$
Remoção da cauda	$\Theta(1)$
Remoção de uma posição arbitrária	$\Theta(n)$
Acesso à cabeça	$\Theta(1)$
Acesso à cauda	$\Theta(1)$
Acesso à posição arbitrária	$\Theta(n)$

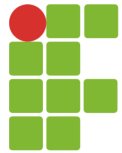


Sumário



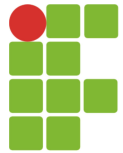
Exemplo da Utilização da Biblioteca

```
8  typedef struct pessoa{
9    char nome[30];
10   char cpf[30];
11   int idade;
12 }pessoa;
```



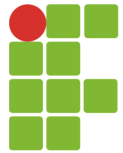
Exemplo da Utilização da Biblioteca

```
14  void* constructor_pessoa(void* data){
15      void* ptr = mallocx(sizeof(pessoa));
16      memcpy(ptr,data,sizeof(pessoa));
17      return ptr;
18  }
```



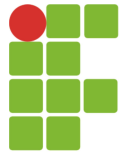
Exemplo da Utilização da Biblioteca

```
20 void destructor_pessoa(void* data){
21   free(data);
22 }
```



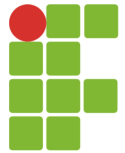
Exemplo da Utilização da Biblioteca

```
25  void* constructor_int(void* data){
26      void* ptr = mallocx(sizeof(int));
27      memcpy(ptr,data,sizeof(int));
28      return ptr;
29  }
```



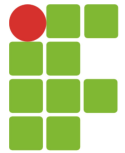
Exemplo da Utilização da Biblioteca

```
31  void destructor_int(void* data){
32      free(data);
33  }
```



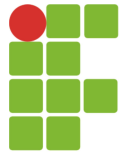
Exemplo da Utilização da Biblioteca

```
35  void print_list_int(list_t* l){
36      printf("\n");
37      list_iterator_t it;
38      for(it=l->head;it!=NULL;it=it->next){
39          printf("%d -> ",*(int*)it->data);
40      }
41      printf("NULL\n");
42      printf("\n");
43  }
```



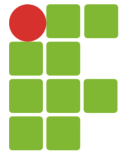
Exemplo da Utilização da Biblioteca

```
45  void imprime_pessoa(pessoa* p){
46    printf("Nome = ");
47    printf("%s\n",p->nome);
48    printf("CPF = ");
49    printf("%s\n",p->cpf);
50    printf("Idade = ");
51    printf("%d\n",p->idade);
52  }
```



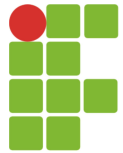
Exemplo da Utilização da Biblioteca

```
56  void print_list_pessoa(list_t* l){
57      printf("\n");
58      list_iterator_t it;
59      for(it=l->head;it!=NULL;it=it->next){
60          imprime_pessoa(it->data);
61      }
62      printf("NULL\n");
63      printf("\n");
64  }
```



Exemplo da Utilização da Biblioteca

```
66  void le_pessoa(pessoa* p){
67    printf("Nome = ");
68    scanf("%s",p->nome);
69    printf("CPF = ");
70    scanf("%s",p->cpf);
71    printf("Idade = ");
72    scanf("%d",&(p->idade));
73  }
```



Exemplo da Utilização da Biblioteca

```
76  int main(){
77      list_t* l1;
78      list_t* l2;
79      list_t* l3;
80      list_initialize(&l1,constructor_pessoa,destructor_pessoa);
81      list_initialize(&l2,constructor_pessoa,destructor_pessoa);
82      list_initialize(&l3,constructor_int,destructor_int);
83      int i;
84      for(i=0;i<3;i++){
85          pessoa p;
86          le_pessoa(&p);
87          list_append(l1,&p);
88          list_prepend(l2,&p);
89          list_append(l3,&i);
90      }
91      print_list_pessoa(l1);
92      print_list_pessoa(l2);
93      print_list_int(l3);
94      list_delete(&l1);
95      list_delete(&l2);
96      list_delete(&l3);
97      return 0;
```