

# Listas

## Estruturas de Dados e Algoritmos



Prof. Daniel Saad Nogueira  
Nunes

IFB – Instituto Federal de Brasília,  
Campus Taguatinga



# Sumário

---

- 1 Introdução
- 2 Listas Encadeadas
- 3 Listas Duplamente Encadeadas
- 4 Exemplos



# Sumário

---

## 1 Introdução



# 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!



# 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

---

## 2 Listas Encadeadas



# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



# Listas Encadeadas: Definição

---

```
7  /**
8   @brief list_node_t Definição de nó de lista ligada.
9   O nó de lista ligada contém um ponteiro para um dado genérico (data)
10  e um ponteiro para o próximo nó da lista.
11  **/
12  typedef struct list_node_t {
13      int data; /*Dado da lista*/
14      struct list_node_t *next; /*ponteiro para o próximo elemento*/
15  } list_node_t;
```



# Listas Encadeadas: Definição

---

```
17  /**
18   @brief list_t Definição do tipo lista. Contém ponteiros para a cabeça e cauda
19   da lista e o tamanho da lista.
20  **/
21  typedef struct list_t {
22      list_node_t *head; /*Cabeça da Lista*/
23      list_node_t *tail; /*Cauda da Lista*/
24      size_t size;      /*tamanho da lista*/
25  } list_t;
```





# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



## Listas Encadeadas: Inicialização

```
38  /** Inicialização dos membros de uma lista **/
39  void list_initialize(list_t **l) {
40      /** Aloca espaço para a estrutura lista **/
41      (*l) = mallocx(sizeof(list_t));
42      /** Cabeça aponta para NULL**/
43      (*l)->head = NULL;
44      /** Cauda aponta para NULL**/
45      (*l)->tail = NULL;
46      /** Tamanho de uma lista recém inicializada é 0 **/
47      (*l)->size = 0;
48  }
```



# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- **Funções auxiliares**
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



## Listas Encadeadas: Funções Auxiliares

---

Retorna o tamanho da lista.

```
264 size_t list_size(list_t *l) {  
265     return l->size;  
266 }
```



## Listas Encadeadas: Funções Auxiliares

---

Retorna verdadeiro se e somente se a lista está vazia.

```
269 bool list_empty(list_t *l) {  
270     return list_size(l) == 0 ? 1 : 0;  
271 }
```



## Listas Encadeadas: Funções Auxiliares

---

Cria um novo nó e o inicializa com um valor.

```
21 static list_node_t *list_new_node(int data) {  
22     /** aloca espaço para novo nó **/  
23     list_node_t *new_node = malloc(sizeof(list_node_t));  
24     /** Constrói o novo dado através da função construtora **/  
25     new_node->data = data;  
26     /** Atribui o ponteiro para o próximo como NULL **/  
27     new_node->next = NULL;  
28     /** Retorna o nó alocado **/  
29     return new_node;  
30 }
```



## Listas Encadeadas: Funções Auxiliares

---

Remove um nó da memória.

```
32  /** Deleta o nó de uma lista **/
33  static void list_delete_node(list_node_t *n) {
34      /** Libera o nó **/
35      free(n);
36  }
```



# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- **Inserção**
- Remoção
- Acesso
- Limpeza
- Análise





# Listas Encadeadas: Inserção

---

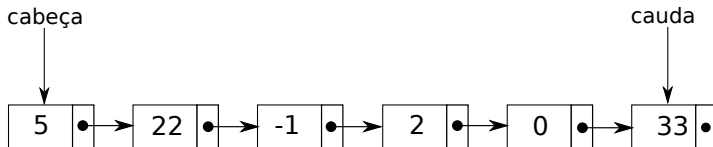
## Inserção

- 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

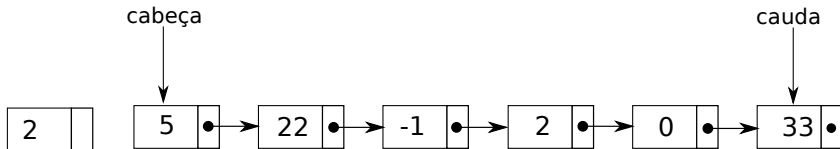
Para inserir na cabeça da lista, criamos um novo nó que aponta para a cabeça atual e depois movemos o ponteiro da cabeça para o novo nó. Se a lista estava vazia, a cauda também deve apontar para o nó recém inserido.





## Listas Encadeadas: Inserção na Cabeça

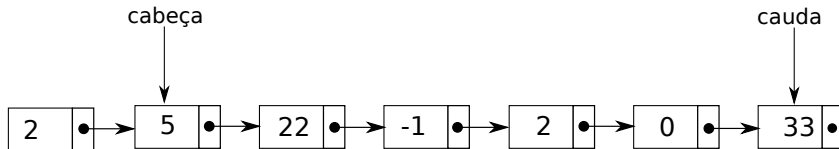
Para inserir na cabeça da lista, criamos um novo nó que aponta para a cabeça atual e depois movemos o ponteiro da cabeça para o novo nó. Se a lista estava vazia, a cauda também deve apontar para o nó recém inserido.





## Listas Encadeadas: Inserção na Cabeça

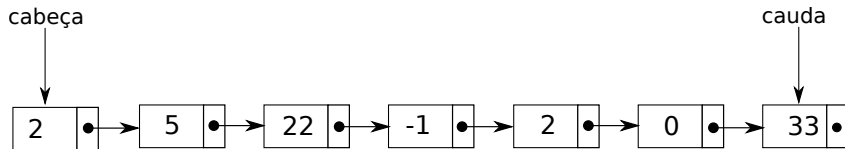
Para inserir na cabeça da lista, criamos um novo nó que aponta para a cabeça atual e depois movemos o ponteiro da cabeça para o novo nó. Se a lista estava vazia, a cauda também deve apontar para o nó recém inserido.





## Listas Encadeadas: Inserção na Cabeça

Para inserir na cabeça da lista, criamos um novo nó que aponta para a cabeça atual e depois movemos o ponteiro da cabeça para o novo nó. Se a lista estava vazia, a cauda também deve apontar para o nó recém inserido.





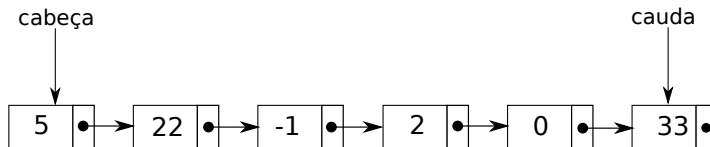
## Listas Encadeadas: Inserção na Cabeça

```
98  /** Insere um elemento na cabeça da lista **/
99  void list_prepend(list_t *l, int data) {
100     /** Cria um novo nó ao invocar list_new_node **/
101     list_node_t *new_node = list_new_node(data);
102     /** Novo nó estabelece uma ligação para a cabeça antiga **/
103     new_node->next = l->head;
104     /** Cabeça antiga aponta agora para o nó recém criado **/
105     l->head = new_node;
106     /** Se a lista estava vazia, a cauda também deve apontar para o nó recém
107         * criado **/
108     if (list_empty(l)) {
109         l->tail = new_node;
110     }
111     /** O tamanho da lista é incrementado **/
112     l->size++;
113 }
```



## Listas Encadeadas: Inserção na Cauda

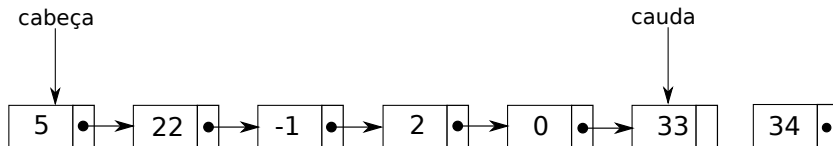
Para inserir um nó na cauda, basta criar um novo nó e fazer com que a cauda aponte para ele. Após isso, a cauda passa a apontar para o nó criado. Caso a lista estivesse vazia, a cabeça também deve apontar para o nó criado





## Listas Encadeadas: Inserção na Cauda

Para inserir um nó na cauda, basta criar um novo nó e fazer com que a cauda aponte para ele. Após isso, a cauda passa a apontar para o nó criado. Caso a lista estivesse vazia, a cabeça também deve apontar para o nó criado

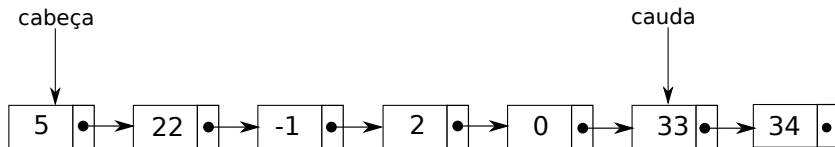






## Listas Encadeadas: Inserção na Cauda

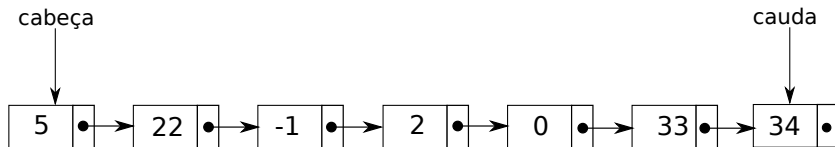
Para inserir um nó na cauda, basta criar um novo nó e fazer com que a cauda aponte para ele. Após isso, a cauda passa a apontar para o nó criado. Caso a lista estivesse vazia, a cabeça também deve apontar para o nó criado





## Listas Encadeadas: Inserção na Cauda

Para inserir um nó na cauda, basta criar um novo nó e fazer com que a cauda aponte para ele. Após isso, a cauda passa a apontar para o nó criado. Caso a lista estivesse vazia, a cabeça também deve apontar para o nó criado





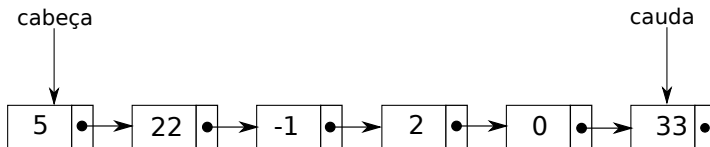
## Listas Encadeadas: Inserção na Cauda

```
115  /** Insere um elemento na cauda da lista **/
116  void list_append(list_t *l, int data) {
117      /** Cria o novo nó ao chamar list_new_node **/
118      list_node_t *new_node = list_new_node(data);
119      /** Se a lista está vazia, a cabeça deve apontar para o nó recém criado **/
120      if (list_empty(l)) {
121          l->head = new_node;
122      }
123      /** Caso contrário, a lista possui uma cauda e ela deve estabelecer
124      * uma ligação o elemento recém criado **/
125      else {
126          l->tail->next = new_node;
127      }
128      /** A cauda é atualizada para apontar para o elemento recém criado **/
129      l->tail = new_node;
130      /** O tamanho da lista é incrementado **/
131      l->size++;
132  }
```



## Listas Encadeadas: Inserção

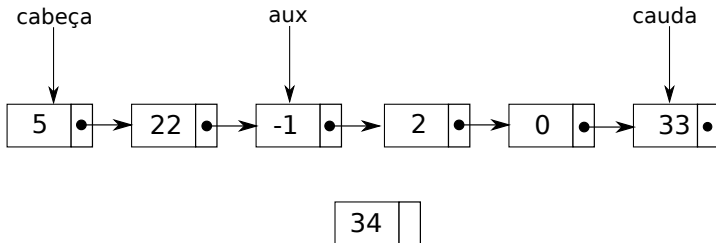
Para inserir em uma posição arbitrária, precisamos percorrer a list até o elemento que antecede a posição de inserção. O novo nó passa a apontar para o nó que sucede este elemento e o elemento passa a apontar para o novo nó.





## Listas Encadeadas: Inserção

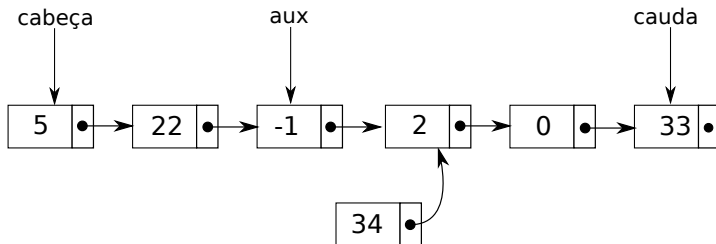
Para inserir em uma posição arbitrária, precisamos percorrer a list até o elemento que antecede a posição de inserção. O novo nó passa a apontar para o nó que sucede este elemento e o elemento passa a apontar para o novo nó.





## Listas Encadeadas: Inserção

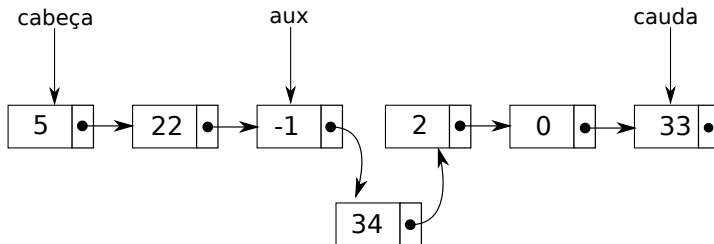
Para inserir em uma posição arbitrária, precisamos percorrer a list até o elemento que antecede a posição de inserção. O novo nó passa a apontar para o nó que sucede este elemento e o elemento passa a apontar para o novo nó.





## Listas Encadeadas: Inserção

Para inserir em uma posição arbitrária, precisamos percorrer a list até o elemento que antecede a posição de inserção. O novo nó passa a apontar para o nó que sucede este elemento e o elemento passa a apontar para o novo nó.





# Listas Encadeadas: Inserção

```
62  /** Insere um novo dado na lista com base na posição i **/
63  void list_insert(list_t *l, int data, size_t i) {
64      /** Apenas modo debug, aborta o programa se a posição for inválida **/
65      assert(i <= list_size(l));
66      /** Se a lista está vazia, ou a posição de inserção é a 0, a
67          inserção é feita na cabeça **/
68      if (list_empty(l) || i == 0) {
69          list_prepend(l, data);
70      }
71      /** Inserção na cauda **/
72      else if (i == list_size(l)) {
73          list_append(l, data);
74      }
```





## Listas Encadeadas: Inserção

```
75  /** Inserção no meio da lista que tem pelo menos 1 elemento **/
76  else {
77      /** Cria o novo nó ao chamar a função list_new_node **/
78      list_node_t *new_node = list_new_node(data);
79      /** Precisamos encontrar o elemento que antecede a posição i ao
80          * caminhar na lista **/
81      list_node_t *aux = l->head;
82      size_t k;
83      /** Caminhamos até a posição i-1 da lista **/
84      for (k = 0; k < i - 1; k++) {
85          aux = aux->next;
86      }
87      /** it agora aponta pro elemento da posição i-1*/
88      /** Estabelecemos o next do novo nó para o elemento antigo da
89          * da posição i **/
90      new_node->next = aux->next;
91      /** O next do nó da posição i-1 recebe o elemento recém inserido **/
92      aux->next = new_node;
93      /** O tamanho da lista é incrementado **/
94      l->size++;
95  }
96 }
```



# Sumário

---

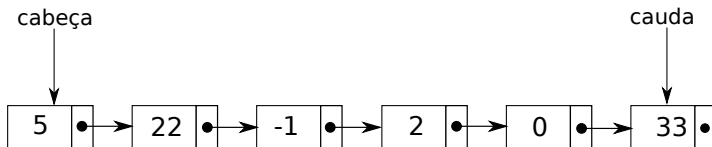
## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- Inserção
- **Remoção**
- Acesso
- Limpeza
- Análise



## Listas Encadeadas: Remoção na Cabeça

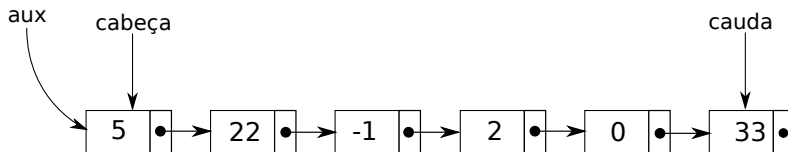
Para remover a cabeça da lista, utilizamos um ponteiro auxiliar que apontará para a cabeça, atualizamos a cabeça para o próximo elemento e removemos o nó apontado pelo ponteiro auxiliar. Caso a lista fique vazia após a remoção deste nó, a cauda deve apontar para NULL.





## Listas Encadeadas: Remoção na Cabeça

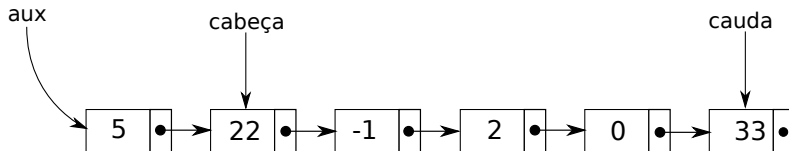
Para remover a cabeça da lista, utilizamos um ponteiro auxiliar que apontará para a cabeça, atualizamos a cabeça para o próximo elemento e removemos o nó apontado pelo ponteiro auxiliar. Caso a lista fique vazia após a remoção deste nó, a cauda deve apontar para NULL.





## Listas Encadeadas: Remoção na Cabeça

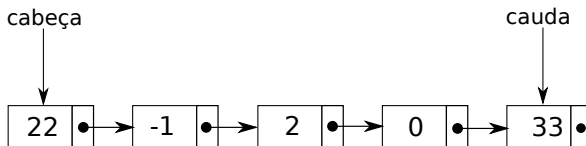
Para remover a cabeça da lista, utilizamos um ponteiro auxiliar que apontará para a cabeça, atualizamos a cabeça para o próximo elemento e removemos o nó apontado pelo ponteiro auxiliar. Caso a lista fique vazia após a remoção deste nó, a cauda deve apontar para NULL.





## Listas Encadeadas: Remoção na Cabeça

Para remover a cabeça da lista, utilizamos um ponteiro auxiliar que apontará para a cabeça, atualizamos a cabeça para o próximo elemento e removemos o nó apontado pelo ponteiro auxiliar. Caso a lista fique vazia após a remoção deste nó, a cauda deve apontar para NULL.





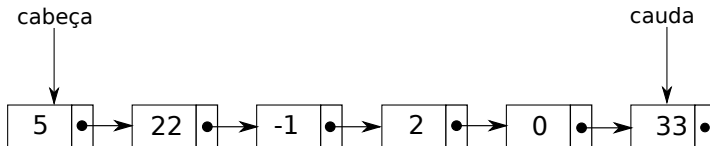
# Listas Encadeadas: Remoção na Cabeça

```
174  /** Remove a cabeça da lista **/
175  void list_remove_head(list_t *l) {
176      /** Debug apenas: aborta o programa caso a remoção da cabeça seja sobre
177       * uma lista vazia **/
178      assert(!list_empty(l));
179      /** O nó a ser removido recebe a cabeça **/
180      list_node_t *aux = l->head;
181      /** Se a lista tem um elemento, após a remoção a cauda deve ser NULL **/
182      if (list_size(l) == 1) {
183          l->tail = NULL;
184      }
185      /** A cabeça passa para o próximo elemento **/
186      l->head = l->head->next;
187      /** Deleta-se a cabeça **/
188      list_delete_node(aux);
189      /** O tamanho da lista é decrementado **/
190      l->size--;
191  }
```



## Listas Encadeadas: Remoção na Cauda

Para remover a cauda, temos que percorrer a lista com um ponteiro auxiliar até chegar ao penúltimo elemento. Assim podemos remover a cauda, fazer com que o nó apontado pelo ponteiro auxiliar aponte para NULL e atualizar a cauda. Caso a lista fique vazia, também deve-se atualizar a cabeça para NULL.

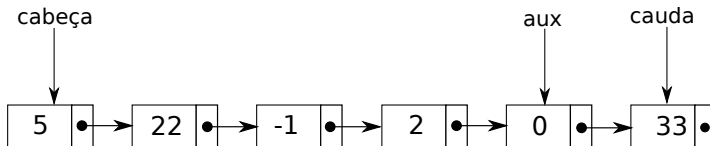






## Listas Encadeadas: Remoção na Cauda

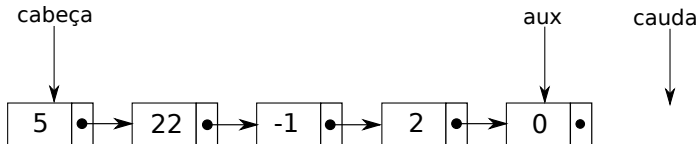
Para remover a cauda, temos que percorrer a lista com um ponteiro auxiliar até chegar ao penúltimo elemento. Assim podemos remover a cauda, fazer com que o nó apontado pelo ponteiro auxiliar aponte para NULL e atualizar a cauda. Caso a lista fique vazia, também deve-se atualizar a cabeça para NULL.





## Listas Encadeadas: Remoção na Cauda

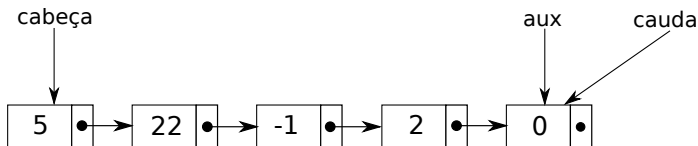
Para remover a cauda, temos que percorrer a lista com um ponteiro auxiliar até chegar ao penúltimo elemento. Assim podemos remover a cauda, fazer com que o nó apontado pelo ponteiro auxiliar aponte para NULL e atualizar a cauda. Caso a lista fique vazia, também deve-se atualizar a cabeça para NULL.





## Listas Encadeadas: Remoção na Cauda

Para remover a cauda, temos que percorrer a lista com um ponteiro auxiliar até chegar ao penúltimo elemento. Assim podemos remover a cauda, fazer com que o nó apontado pelo ponteiro auxiliar aponte para NULL e atualizar a cauda. Caso a lista fique vazia, também deve-se atualizar a cabeça para NULL.





## Listas Encadeadas: Remoção na Cauda

```
193  /** Remove a cauda da lista */
194  void list_remove_tail(list_t *l) {
195      /** Debug apenas, aborta o programa caso a função seja chamada para uma
196       * lista vazia */
197      assert(list_size(l) > 0);
198      /** O nó a ser removido recebe a cauda */
199      list_node_t *to_be_removed = l->tail;
200      /** Se a lista tem tamanho 1, a cauda e a cabeça apontam para NULL
201       * após a remoção */
202      if (list_size(l) == 1) {
203          l->head = NULL;
204          l->tail = NULL;
205      }
```



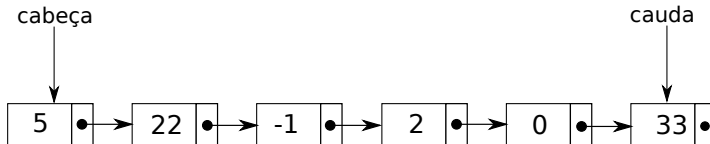
## Listas Encadeadas: Remoção na Cauda

```
206  /** Caso contrário, a lista tem mais de um elemento. Deve-se iterar sobre
207  * a lista até o penúltimo elemento **/
208  else {
209      /** Itera-se sobre a lista a partir da cabeça até o penúltimo elemento
210      **/
211      list_node_t *aux = l->head;
212      while (aux->next != l->tail) {
213          aux = aux->next;
214      }
215      /** O campo next do penúltimo elemento agora aponta para NULL **/
216      aux->next = NULL;
217      /** O penúltimo elemento passa a ser a cauda **/
218      l->tail = aux;
219  }
220  /** Remove-se a cauda antiga **/
221  list_delete_node(to_be_removed);
222  /** O tamanho da lista é decrementado **/
223  l->size--;
224  }
```



## Listas Encadeadas: Remoção

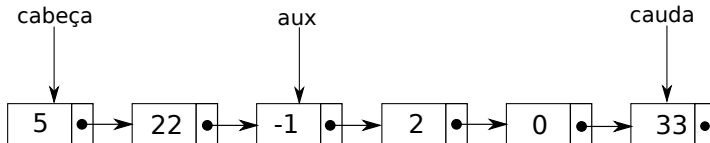
Para remover um elemento arbitrário, devemos percorrer a lista até o nó anterior a posição de remoção para que ele aponte para o nó que sucede o nó a ser removido. Em seguida, removemos o nó.





## Listas Encadeadas: Remoção

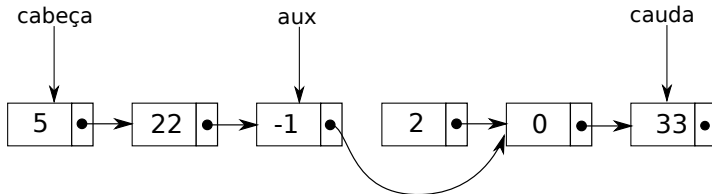
Para remover um elemento arbitrário, devemos percorrer a lista até o nó anterior a posição de remoção para que ele aponte para o nó que sucede o nó a ser removido. Em seguida, removemos o nó.





## Listas Encadeadas: Remoção

Para remover um elemento arbitrário, devemos percorrer a lista até o nó anterior a posição de remoção para que ele aponte para o nó que sucede o nó a ser removido. Em seguida, removemos o nó.

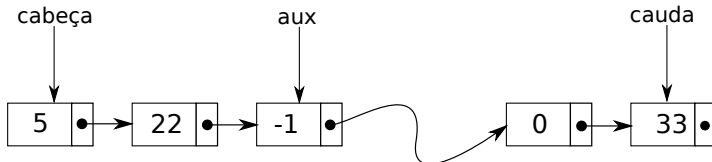






## Listas Encadeadas: Remoção

Para remover um elemento arbitrário, devemos percorrer a lista até o nó anterior a posição de remoção para que ele aponte para o nó que sucede o nó a ser removido. Em seguida, removemos o nó.





## Listas Encadeadas: Remoção

```
134  /** Remove o elemento da posição i da lista **/
135  void list_remove(list_t *l, size_t i) {
136      /** Debug apenas, aborta o programa se a remoção estiver sendo feita
137      * em uma lista vazia ou em uma posição inexistente da lista **/
138      assert(!list_empty(l) && i < list_size(l));
139      /** Se a lista tem tamanho 1, ou a remoção é do primeiro elemento,
140      equivale a eliminar a cabeça
141      **/
142      if (list_size(l) == 1 || i == 0) {
143          list_remove_head(l);
144      }
145      /** Se i==list_size(l)-1, a remoção é na cauda **/
146      else if (i == list_size(l) - 1) {
147          list_remove_tail(l);
148      }
149      /** O nó a ser removido encontra-se no meio da lista e a lista
150      possuir mais que um elemento **/
```



## Listas Encadeadas: Remoção

```
151     else {
152         /** Nó a ser removido **/
153         list_node_t *node;
154         /** Devemos percorrer até o i-1-ésimo elemento a partir
155             * da cabeça **/
156         list_node_t *aux = l->head;
157         size_t k;
158         /** Itera-se até o elemento imediatamente interior ao elemento i **/
159         for (k = 0; k < i - 1; k++) {
160             aux = aux->next;
161         }
162         /** Nó a ser removido passa a ser o i-ésimo elemento **/
163         node = aux->next;
164         /** O anterior ao nó a ser removido aponta para o elemento
165             * que vem após o nó a ser removido **/
166         aux->next = node->next;
167         /** Deleta o nó atribuído anteriormente **/
168         list_delete_node(node);
169         /** Decrementa o tamanho da lista **/
170         l->size--;
171     }
172 }
```



# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- Inserção
- Remoção
- **Acesso**
- Limpeza
- Análise



## 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

---

Acesso a cabeça da lista.

```
249  /** Acessa a cabeça da lista */
250  int list_access_head(list_t *l) {
251      /** Debug apenas, aborta o programa se a lista não tem cabeça (é vazia) */
252      assert(!(list_empty(l)));
253      return (l->head->data);
254  }
```



## Listas Encadeadas: Acesso

---

Acesso a cauda.

```
256  /** Acessa a cauda da lista */
257  int list_access_tail(list_t *l) {
258      /** Debug apenas, aborta o programa se a lista não tem cauda (é vazia) */
259      assert(!list_empty(l));
260      return (l->tail->data);
261  }
```



## Listas Encadeadas: Acesso

---

Acesso a uma posição arbitrária.

```
226  /** Acessa o i-ésimo elemento da lista **/
227  int list_access(list_t *l, size_t i) {
228      /** Debug apenas, aborta o programa em caso de posição inválida
229          * a ser acessada **/
230      assert(!list_empty(l) && i < list_size(l));
231      /** Se i==0, o acesso é na cabeça **/
232      if (i == 0) {
233          return (list_access_head(l));
234      }
235      /** Se i==list_size(l)-1, o acesso é na cauda **/
236      else if (i == list_size(l) - 1) {
237          return (list_access_tail(l));
238      }
```





## Listas Encadeadas: Acesso

---

Acesso a uma posição arbitrária.

```
239  /** Caso contrário, percorre-se a lista até o i-ésimo elemento **/
240  size_t j;
241  list_node_t *it = l->head;
242  for (j = 0; j < i; j++) {
243      it = it->next;
244  }
245  /** O campo dado do elemento acessado é retornado **/
246  return (it->data);
247 }
```



# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- Inserção
- Remoção
- Acesso
- **Limpeza**
- Análise



## 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

---

```
50  /** Deleta uma lista com sucessivas remoções da cabeça **/
51  void list_delete(list_t **l) {
52      /** Enquanto a lista não for vazia, remove a cabeça **/
53      while (!list_empty(*l)) {
54          list_remove_head(*l);
55      }
56      /** Desaloca espaço para a estrutura de dados **/
57      free(*l);
58      /** Atribui o valor NULL ao ponteiro da lista **/
59      *l = NULL;
60  }
```



# Sumário

---

## 2 Listas Encadeadas

- Definição
- Inicialização
- Funções auxiliares
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



# 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

---

## 3 Listas Duplamente Encadeadas



# 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

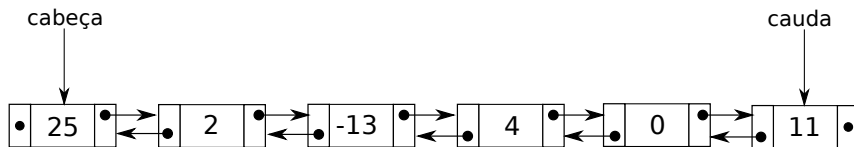
---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



# Listas Duplamente Encadeadas





## Listas Duplamente Encadeadas: Definição

---

```
7  /**A nossa dlista encadeada consiste de vários nós,  
8  que possuem o tipo dlist_node_t */  
9  typedef struct dlist_node_t {  
10     int data; /*Dado*/  
11     struct dlist_node_t *next; /*ponteiro para o próximo elemento*/  
12     struct dlist_node_t *prev; /*Ponteiro para o elemento anterior*/  
13 } dlist_node_t;
```



# Listas Duplamente Encadeadas: Definição

---

```
15 typedef struct dlist_t {  
16     dlist_node_t *head; /*Cabeça da dlista*/  
17     dlist_node_t *tail; /*Cauda da dlista*/  
18     size_t size;        /*tamanho da dlista*/  
19 } dlist_t;
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



## Listas Duplamente Encadeadas: Inicialização

---

```
20  /**Inicializa a lista duplamente encadeada e seus membros**/
21  void dlist_initialize(dlist_t **l) {
22      (*l) = mallocx(sizeof(dlist_t));
23      (*l)->head = NULL;
24      (*l)->tail = NULL;
25      (*l)->size = 0;
26  }
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- **Funções Auxiliares**
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



## Listas Duplamente Encadeadas: Funções Auxiliares

---

```
155  /**Retorna o tamanho da dlista**/
156  size_t dlist_size(dlist_t *l) {
157      return l->size;
158  }
```





## Listas Duplamente Encadeadas: Funções Auxiliares

---

```
160  /**Retorna verdadeiro se a dlista está vazia, e falso caso contrário**/
161  bool dlist_empty(dlist_t *l) {
162      return dlist_size(l) == 0 ? true : false;
163  }
```



## Listas Duplamente Encadeadas: Funções Auxiliares

---

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



## Listas Duplamente Encadeadas: Funções Auxiliares

---

```
16 static void dlist_delete_node(dlist_node_t *node) {  
17     free(node);  
18 }
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- **Inserção**
- Remoção
- Acesso
- Limpeza
- Análise



# 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

```
61  /** Insere um elemento na cabeça da dlista **/
62  void dlist_prepend(dlist_t *l, int data) {
63      dlist_node_t *new_node = dlist_new_node(data);
64      if (dlist_empty(l)) {
65          l->tail = new_node;
66      } else {
67          l->head->prev = new_node;
68      }
69      new_node->next = l->head;
70      l->head = new_node;
71      l->size++;
72  }
```



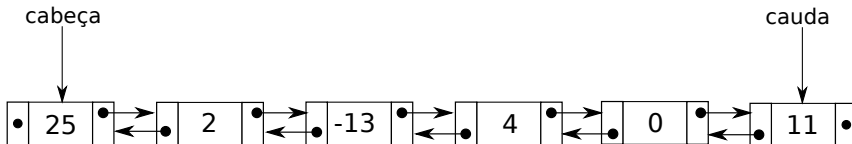
## Listas Duplamente Encadeadas: Inserção na Cauda

```
74  /**Inserir um elemento na cauda da dlista **/
75  void dlist_append(dlist_t *l, int data) {
76      dlist_node_t *new_node = dlist_new_node(data);
77      if (dlist_empty(l)) {
78          l->head = new_node;
79      } else {
80          l->tail->next = new_node;
81      }
82      new_node->prev = l->tail;
83      l->tail = new_node;
84      l->size++;
85  }
```



## Listas Duplamente Encadeadas: Inserção

Percorremos a lista até chegar na posição que antecede a inserção para conseguirmos encaixar o novo nó entre dois nós existentes.

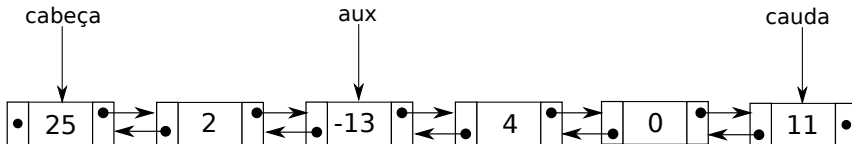






## Listas Duplamente Encadeadas: Inserção

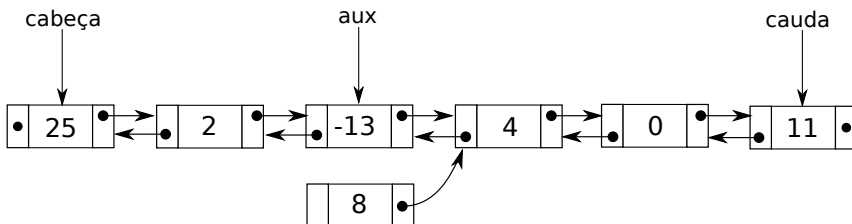
Percorremos a lista até chegar na posição que antecede a inserção para conseguirmos encaixar o novo nó entre dois nós existentes.





## Listas Duplamente Encadeadas: Inserção

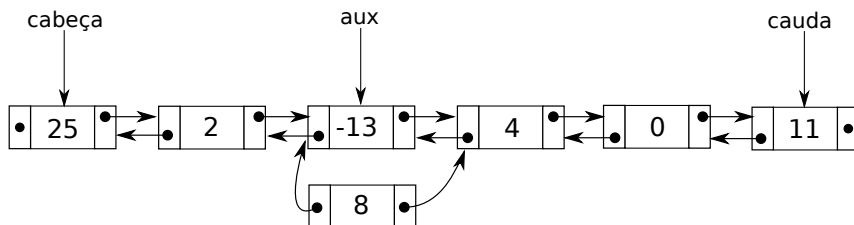
Percorremos a lista até chegar na posição que antecede a inserção para conseguirmos encaixar o novo nó entre dois nós existentes.





## Listas Duplamente Encadeadas: Inserção

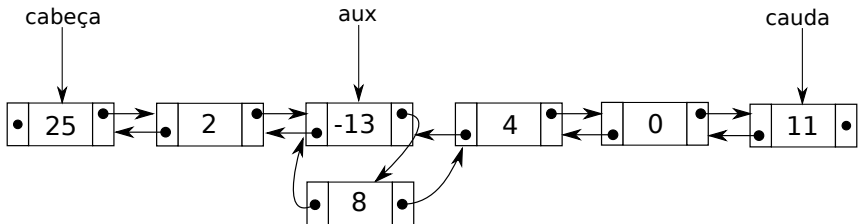
Percorremos a lista até chegar na posição que antecede a inserção para conseguirmos encaixar o novo nó entre dois nós existentes.





## Listas Duplamente Encadeadas: Inserção

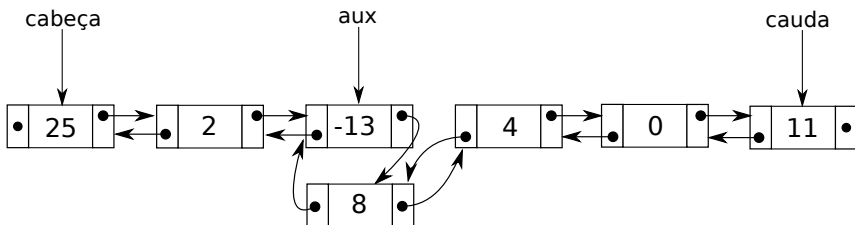
Percorremos a lista até chegar na posição que antecede a inserção para conseguirmos encaixar o novo nó entre dois nós existentes.





## Listas Duplamente Encadeadas: Inserção

Percorremos a lista até chegar na posição que antecede a inserção para conseguirmos encaixar o novo nó entre dois nós existentes.





## Listas Duplamente Encadeadas: Inserção

```
36 void dlist_insert(dlist_t *l, int data, size_t i) {
37     assert(i <= dlist_size(l));
38     if (dlist_empty(l) || i == 0) {
39         dlist_prepend(l, data);
40     } else if (i == dlist_size(l)) {
41         /*Inserção na cauda*/
42         dlist_append(l, data);
43     } else {
44         dlist_node_t *new_node = dlist_new_node(data);
45         /*Inserção no meio da lista*/
46         /*Precisamos encontrar o elemento que antecede a posição i*/
47         dlist_node_t *aux = l->head;
48         size_t k;
49         for (k = 0; k < i - 1; k++) {
50             aux = aux->next;
51         }
52         /*aux agora aponta pro elemento da posição i-1*/
53         new_node->next = aux->next;
54         new_node->prev = aux;
55         aux->next->prev = new_node;
56         aux->next = new_node;
57         l->size++;
58     }
59 }
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- Inserção
- **Remoção**
- Acesso
- Limpeza
- Análise



## 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

```
109 void dlist_remove_head(dlist_t *l) {
110     assert(!dlist_empty(l));
111     dlist_node_t *node = l->head;
112     l->head = l->head->next;
113     if (dlist_size(l) == 1) {
114         l->tail = NULL;
115     } else {
116         l->head->prev = NULL;
117     }
118     dlist_delete_node(node);
119     l->size--;
120 }
```



## 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.



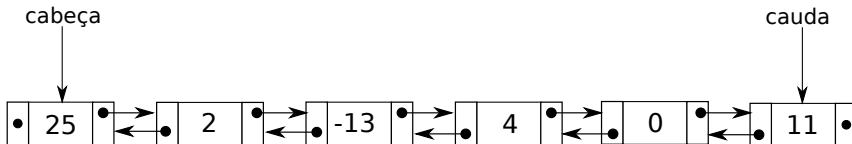
## Listas Duplamente Encadeadas: Remoção na Cauda

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



## Listas Duplamente Encadeadas: Remoção

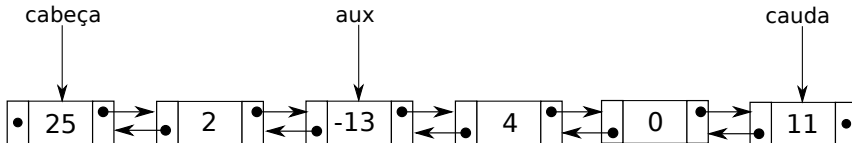
Para remoção em uma posição arbitrária, percorremos a lista até chegar na posição que queremos remover.





## Listas Duplamente Encadeadas: Remoção

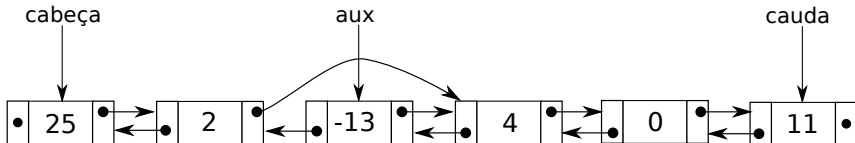
Para remoção em uma posição arbitrária, percorremos a lista até chegar na posição que queremos remover.





## Listas Duplamente Encadeadas: Remoção

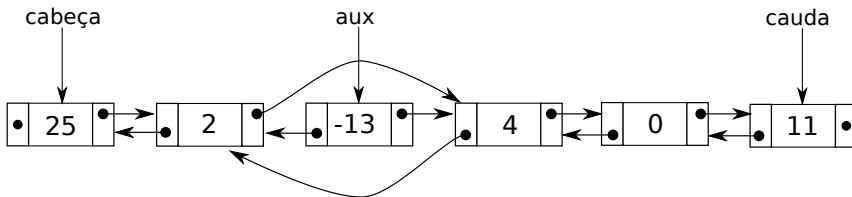
Para remoção em uma posição arbitrária, percorremos a lista até chegar na posição que queremos remover.





## Listas Duplamente Encadeadas: Remoção

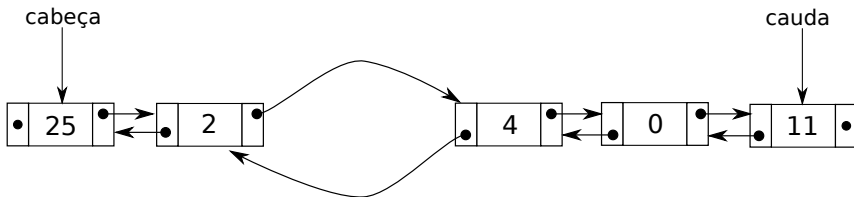
Para remoção em uma posição arbitrária, percorremos a lista até chegar na posição que queremos remover.





## Listas Duplamente Encadeadas: Remoção

Para remoção em uma posição arbitrária, percorremos a lista até chegar na posição que queremos remover.







## Listas Duplamente Encadeadas: Inserção

```
87  /**Remove o elemento da posição i da dlista**/
88  void dlist_remove(dlist_t *l, size_t i) {
89      assert(!dlist_empty(l) && i < dlist_size(l));
90      dlist_node_t *node;
91      if (dlist_size(l) == 1 || i == 0) {
92          dlist_remove_head(l);
93      } else if (i == dlist_size(l) - 1) {
94          dlist_remove_tail(l);
95      } else {
96          dlist_node_t *aux = l->head;
97          size_t k;
98          for (k = 0; k < i; k++) {
99              aux = aux->next;
100          }
101          node = aux;
102          node->prev->next = node->next;
103          node->next->prev = node->prev;
104          dlist_delete_node(node);
105          l->size--;
106      }
107  }
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- Inserção
- Remoção
- **Acesso**
- Limpeza
- Análise



## 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

---

```
145 int dlist_access_head(dlist_t *l) {  
146     assert(!(dlist_empty(l)));  
147     return (l->head->data);  
148 }
```



## Listas Duplamente Encadeadas: Acesso

---

```
148 }  
149  
150 int dlist_access_tail(dlist_t *l) {  
151     assert(!dlist_empty(l));  
152     return (l->tail->data);  
153 }
```



## Listas Duplamente Encadeadas: Acesso

```
135 int dlist_access(dlist_t *l, size_t i) {  
136     assert(!dlist_empty(l) && i < dlist_size(l));  
137     dlist_node_t *aux = l->head;  
138     int j;  
139     for (j = 0; j < i; j++) {  
140         aux = aux->next;  
141     }  
142     return (aux->data);  
143 }
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- Inserção
- Remoção
- Acesso
- **Limpeza**
- Análise



## Listas Duplamente Encadeadas: Limpeza

---

- Funciona de forma análoga ao das listas encadeadas simples.





## Listas Duplamente Encadeadas: Limpeza

---

```
28 void dlist_delete(dlist_t **l) {  
29     while (!dlist_empty(*l)) {  
30         dlist_remove_head(*l);  
31     }  
32     free(*l);  
33     *l = NULL;  
34 }
```



# Sumário

---

## 3 Listas Duplamente Encadeadas

- Definição
- Inicialização
- Funções Auxiliares
- Inserção
- Remoção
- Acesso
- Limpeza
- Análise



# 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

---

## 4 Exemplos



## Exemplo da Utilização da Biblioteca

---

```
1  #include "list.h"
2  #include <stdio.h>
3  #include <string.h>
4
5  void print_list(list_t *l) {
6      printf("\n");
7      list_node_t *aux;
8      for (aux = l->head; aux != NULL; aux = aux->next) {
9          printf("%d -> ", aux->data);
10     }
11     printf("NULL\n");
12     printf("\n");
13 }
```



## Exemplo da Utilização da Biblioteca

```
14
15 int main() {
16     list_t *l;
17     list_initialize(&l);
18     int i;
19     for (i = 0; i < 3; i++) {
20         list_append(l, i);
21     }
22     print_list(l);
23     list_insert(l, 5, 1);
24     print_list(l);
25     list_remove_head(l);
26     list_remove_tail(l);
27     print_list(l);
28     list_delete(&l);
29     return 0;
30 }
```