MC-202 — Unidade 7
Operações em listas e variações

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

1º semestre/2018

Operações em lista ligada

Vamos ver três novas operações para listas ligadas

Operações em lista ligada

Vamos ver três novas operações para listas ligadas

```
1 typedef struct No {
2   int dado;
3   struct No *prox;
4 } No;
5
6 typedef struct No * p_no;
7
8 p_no criar_lista();
9 void destruir_lista(p_no lista);
10 p_no adicionar_elemento(p_no lista, int x);
11 void imprime(p_no lista);
```

Operações em lista ligada

Vamos ver três novas operações para listas ligadas

```
1  typedef struct No {
2    int dado;
3    struct No *prox;
4  } No;
5
6  typedef struct No * p_no;
7
8  p_no criar_lista();
9  void destruir_lista(p_no lista);
10 p_no adicionar_elemento(p_no lista, int x);
11 void imprime(p_no lista);
```

Operações em lista ligada

Vamos ver três novas operações para listas ligadas

```
1  typedef struct No {
2    int dado;
3    struct No *prox;
4  } No;
5
6  typedef struct No * p_no;
7
8  p_no criar_lista();
9  void destruir_lista(p_no lista);
10 p_no adicionar_elemento(p_no lista, int x);
11 void imprime(p_no lista);
12
13 p_no copiar_lista(p_no lista);
```

Operações em lista ligada

Vamos ver três novas operações para listas ligadas

```
1  typedef struct No {
2    int dado;
3    struct No *prox;
4  } No;
5
6  typedef struct No * p_no;
7
8  p_no criar_lista();
9  void destruir_lista(p_no lista);
10 p_no adicionar_elemento(p_no lista, int x);
11 void imprime(p_no lista);
12
13 p_no copiar_lista(p_no lista);
14 p_no inverter_lista(p_no lista);
```

Operações em lista ligada

Vamos ver três novas operações para listas ligadas

```
1  typedef struct No {
2    int dado;
3    struct No *prox;
4  } No;
5
6  typedef struct No * p_no;
7
8  p_no criar_lista();
9  void destruir_lista(p_no lista);
10 p_no adicionar_elemento(p_no lista, int x);
11 void imprime(p_no lista);
12
13 p_no copiar_lista(p_no lista);
14 p_no inverter_lista(p_no lista);
15 p_no concatenar_lista(p_no primeira, p_no segunda);
```

Copiando

Versão recursiva:

Copiando

Versão recursiva:

```
1 p_no copiar_lista(p_no lista) {
```

Copiando

Versão recursiva:

```
1 p_no copiar_lista(p_no lista) {
```

Copiando

Versão recursiva:

```
1 p_no copiar_lista(p_no lista) {
2   p_no novo;
3   if(lista == NULL)
4     return NULL;
```

Copiando

Versão recursiva:

```
1 p_no copiar_lista(p_no lista) {
2   p_no novo;
3   if(lista == NULL)
4     return NULL;
5   novo = malloc(sizeof(No));
6   novo->dado = lista->dado;
7   novo->prox = copiar_lista(lista->prox);
```

Copiando

Versão recursiva:

```
1 p_no copiar_lista(p_no lista) {
2   p_no novo;
3   if(lista == NULL)
4     return NULL;
5   novo = malloc(sizeof(No));
6   novo->dado = lista->dado;
7   novo->prox = copiar_lista(lista->prox);
8   return novo;
9 }
```

Copiando

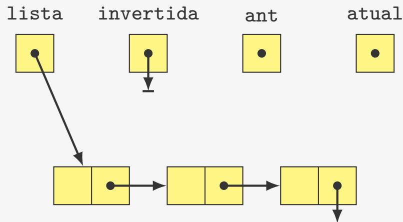
Versão recursiva:

```
1 p_no copiar_lista(p_no lista) {
2   p_no novo;
3   if(lista == NULL)
4     return NULL;
5   novo = malloc(sizeof(No));
6   novo->dado = lista->dado;
7   novo->prox = copiar_lista(lista->prox);
8   return novo;
9 }
```

Exercício: implemente uma versão iterativa da função

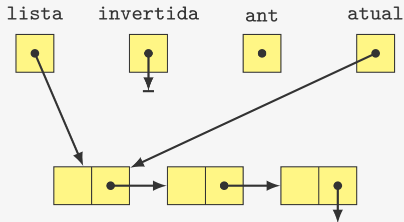
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



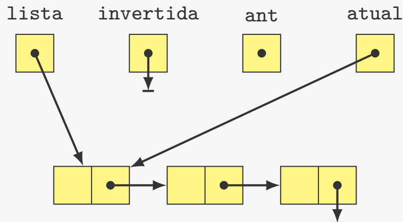
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



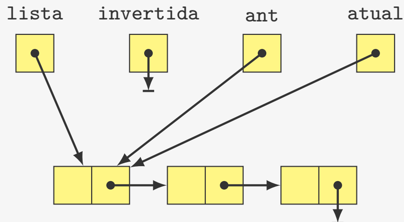
Invertendo

```
1 p_no inverter_lista(p_no lista) {
2   p_no atual, ant, invertida = NULL;
3   atual = lista;
4   while (atual != NULL) {
5     ant = atual;
6     atual = ant->prox;
7     ant->prox = invertida;
8     invertida = ant;
9   }
10  return invertida;
11 }
```



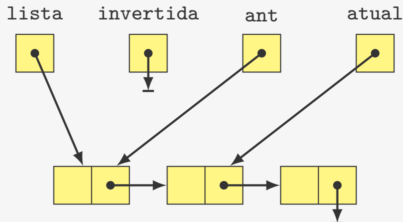
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



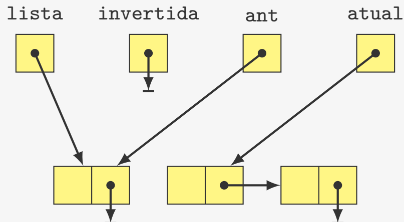
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



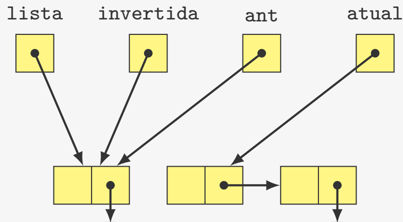
Invertendo

```
1 p_no inverter_lista(p_no lista) {
2   p_no atual, ant, invertida = NULL;
3   atual = lista;
4   while (atual != NULL) {
5     ant = atual;
6     atual = ant->prox;
7     ant->prox = invertida;
8     invertida = ant;
9   }
10  return invertida;
11 }
```



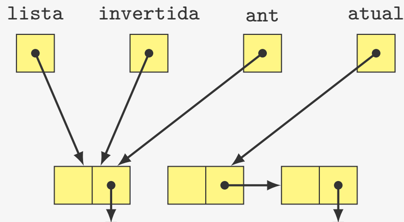
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



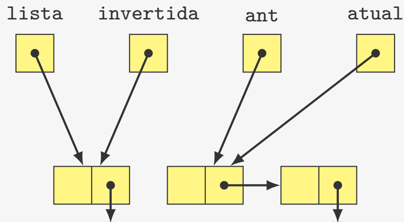
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



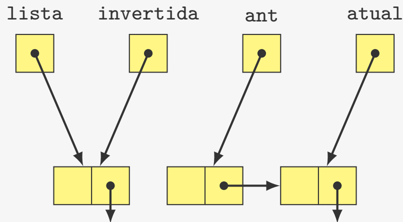
Invertendo

```
1 p_no inverter_lista(p_no lista) {
2   p_no atual, ant, invertida = NULL;
3   atual = lista;
4   while (atual != NULL) {
5     ant = atual;
6     atual = ant->prox;
7     ant->prox = invertida;
8     invertida = ant;
9   }
10  return invertida;
11 }
```



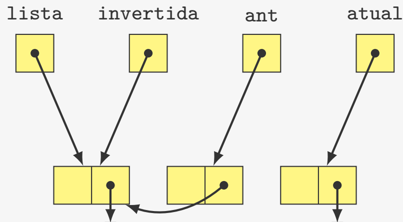
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



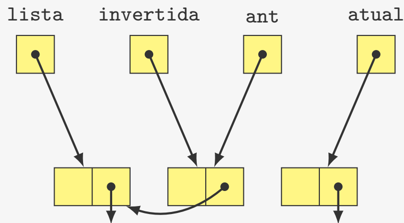
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



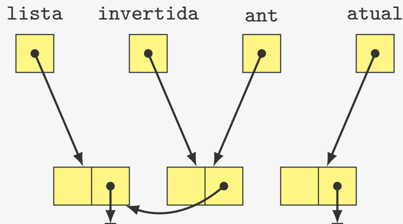
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



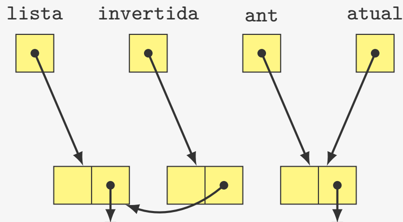
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



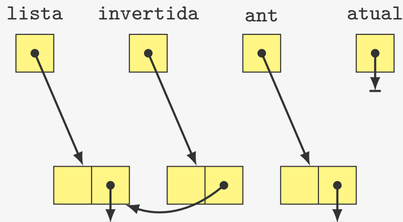
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



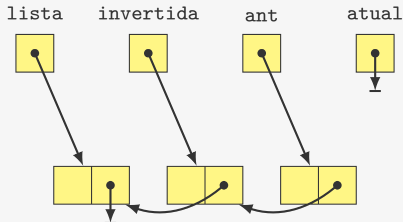
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



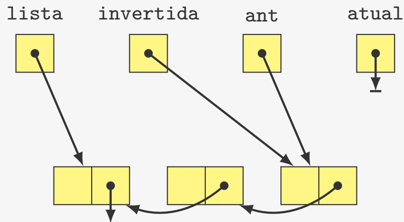
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



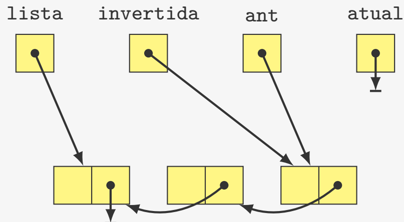
Invertendo

```
1  p_no inverter_lista(p_no lista) {
2    p_no atual, ant, invertida = NULL;
3    atual = lista;
4    while (atual != NULL) {
5      ant = atual;
6      atual = ant->prox;
7      ant->prox = invertida;
8      invertida = ant;
9    }
10   return invertida;
11 }
```



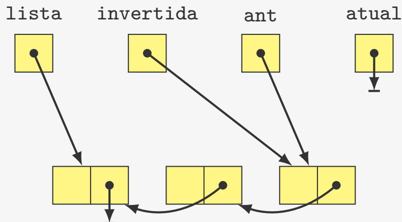
Invertendo

```
1 p_no inverter_lista(p_no lista) {
2   p_no atual, ant, invertida = NULL;
3   atual = lista;
4   while (atual != NULL) {
5     ant = atual;
6     atual = ant->prox;
7     ant->prox = invertida;
8     invertida = ant;
9   }
10  return invertida;
11 }
```



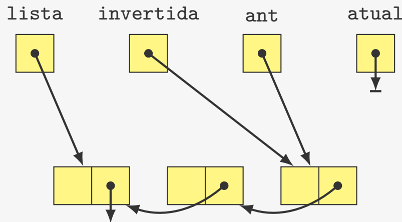
Invertendo

```
1 p_no inverter_lista(p_no lista) {
2   p_no atual, ant, invertida = NULL;
3   atual = lista;
4   while (atual != NULL) {
5     ant = atual;
6     atual = ant->prox;
7     ant->prox = invertida;
8     invertida = ant;
9   }
10  return invertida;
11 }
```



Invertendo

```
1 p_no inverter_lista(p_no lista) {
2   p_no atual, ant, invertida = NULL;
3   atual = lista;
4   while (atual != NULL) {
5     ant = atual;
6     atual = ant->prox;
7     ant->prox = invertida;
8     invertida = ant;
9   }
10  return invertida;
11 }
```



Exercício: implemente uma versão recursiva da função

Concatenando

```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
```

Concatenando

```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
```

Concatenando

```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
2   if (primeira == NULL)
```

Concatenando

```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
2   if (primeira == NULL)
3     return segunda;
```

Concatenando

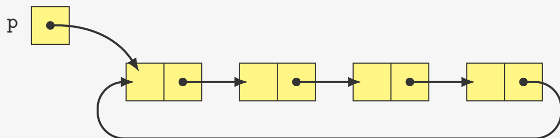
```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
2   if (primeira == NULL)
3     return segunda;
4   primeira->prox = concatenar_lista(primeira->prox, segunda);
```

Concatenando

```
1 p_no concatenar_lista(p_no primeira, p_no segunda) {
2   if (primeira == NULL)
3     return segunda;
4   primeira->prox = concatenar_lista(primeira->prox, segunda);
5   return primeira;
6 }
```

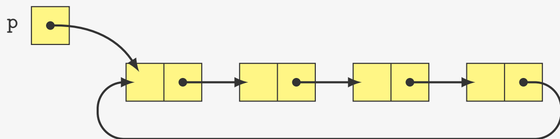
Variações - Listas circulares

Lista circular:

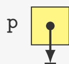


Variações - Listas circulares

Lista circular:

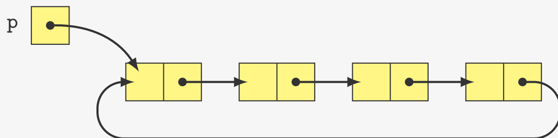


Lista circular vazia:

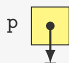


Variações - Listas circulares

Lista circular:



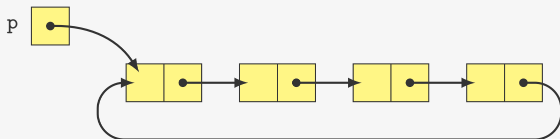
Lista circular vazia:



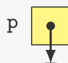
Exemplo de aplicações:

Variações - Listas circulares

Lista circular:



Lista circular vazia:

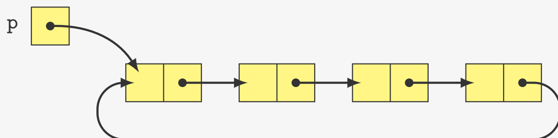


Exemplo de aplicações:

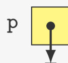
- Execução de processos no sistema operacional

Variações - Listas circulares

Lista circular:



Lista circular vazia:



Exemplo de aplicações:

- Execução de processos no sistema operacional
- Controlar de quem é a vez em um jogo de tabuleiro

Inserindo em lista circular

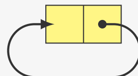
```
1 p_no inserir_circular(p_no lista, int x) {
```

Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
```

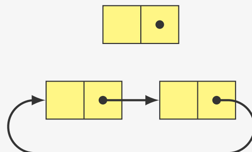
Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
```



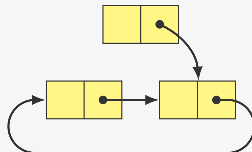
Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
```



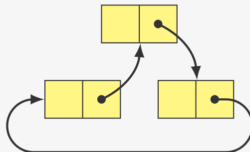
Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
8     novo->prox = lista->prox;
```



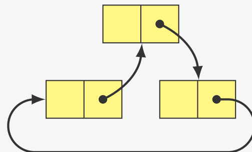
Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
8     novo->prox = lista->prox;
9     lista->prox = novo;
10  }
```



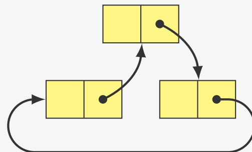
Inserindo em lista circular

```
1  p_no inserir_circular(p_no lista, int x) {
2    p_no novo;
3    novo = malloc(sizeof(No));
4    novo->dado = x;
5    if (lista == NULL)
6      novo->prox = novo;
7    else {
8      novo->prox = lista->prox;
9      lista->prox = novo;
10   }
11   return novo;
12 }
```



Inserindo em lista circular

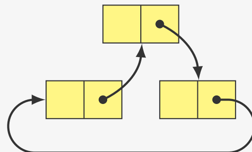
```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
8     novo->prox = lista->prox;
9     lista->prox = novo;
10  }
11  return novo;
12 }
```



Observações:

Inserindo em lista circular

```c
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
8     novo->prox = lista->prox;
9     lista->prox = novo;
10  }
11  return novo;
12 }
```

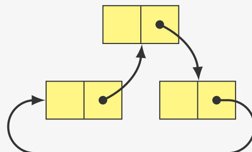


Observações:

- O elemento é inserido na segunda posição

Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
8     novo->prox = lista->prox;
9     lista->prox = novo;
10  }
11  return novo;
12 }
```

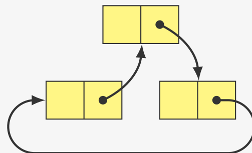


Observações:

- O elemento é inserido na segunda posição
  - Para inserir na primeira precisaria percorrer a lista... $O(n)$

Inserindo em lista circular

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   if (lista == NULL)
6     novo->prox = novo;
7   else {
8     novo->prox = lista->prox;
9     lista->prox = novo;
10  }
11  return novo;
12 }
```

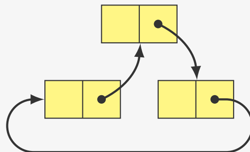


Observações:

- O elemento é inserido na segunda posição
  - Para inserir na primeira precisaria percorrer a lista... $O(n)$
- É devolvido o ponteiro para o novo elemento

Inserindo em lista circular

```c
p_no inserir_circular(p_no lista, int x) {
  p_no novo;
  novo = malloc(sizeof(No));
  novo->dado = x;
  if (lista == NULL)
    novo->prox = novo;
  else {
    novo->prox = lista->prox;
    lista->prox = novo;
  }
  return novo;
}
```



Observações:

- O elemento é inserido na segunda posição
  - Para inserir na primeira precisaria percorrer a lista... $\mathrm{O}(n)$
- É devolvido o ponteiro para o novo elemento
  - Ex: ao inserir 0 em $(3, 7, 2)$ ficamos com $(0, 7, 2, 3)$

Removendo de lista circular

```
1 p_no remover_circular(p_no lista, p_no no) {
```

Removendo de lista circular

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
```

Removendo de lista circular

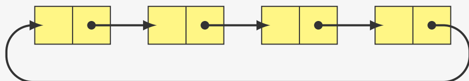
```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
4     free(no);
5     return NULL;
6   }
```

Removendo de lista circular

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
4     free(no);
5     return NULL;
6   }
7   for(ant = no->prox; ant->prox != no; ant = ant->prox);
```

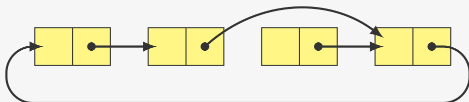
Removendo de lista circular

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
4     free(no);
5     return NULL;
6   }
7   for(ant = no->prox; ant->prox != no; ant = ant->prox);
8   ant->prox = no->prox;
```



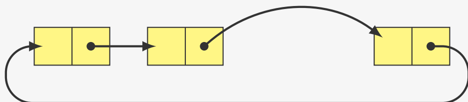
Removendo de lista circular

```
1  p_no remover_circular(p_no lista, p_no no) {
2    p_no ant;
3    if (no->prox == no) {
4      free(no);
5      return NULL;
6    }
7    for(ant = no->prox; ant->prox != no; ant = ant->prox);
8    ant->prox = no->prox;
9    if (lista == no)
10     lista = lista->prox;
```



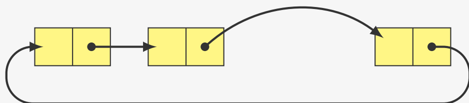
Removendo de lista circular

```
 1 p_no remover_circular(p_no lista, p_no no) {
 2   p_no ant;
 3   if (no->prox == no) {
 4     free(no);
 5     return NULL;
 6   }
 7   for(ant = no->prox; ant->prox != no; ant = ant->prox);
 8   ant->prox = no->prox;
 9   if (lista == no)
10     lista = lista->prox;
11   free(no);
```



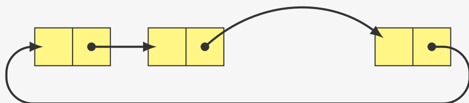
Removendo de lista circular

```
1  p_no remover_circular(p_no lista, p_no no) {
2    p_no ant;
3    if (no->prox == no) {
4      free(no);
5      return NULL;
6    }
7    for(ant = no->prox; ant->prox != no; ant = ant->prox);
8    ant->prox = no->prox;
9    if (lista == no)
10     lista = lista->prox;
11   free(no);
12   return lista;
13 }
```



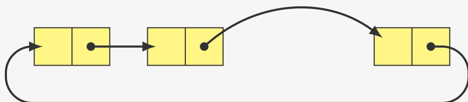
Removendo de lista circular

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
4     free(no);
5     return NULL;
6   }
7   for(ant = no->prox; ant->prox != no; ant = ant->prox);
8   ant->prox = no->prox;
9   if (lista == no)
10    lista = lista->prox;
11  free(no);
12  return lista;
13 }
```



Removendo de lista circular

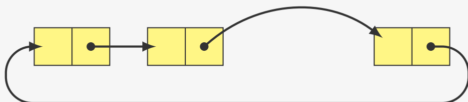
```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
4     free(no);
5     return NULL;
6   }
7   for(ant = no->prox; ant->prox != no; ant = ant->prox);
8   ant->prox = no->prox;
9   if (lista == no)
10    lista = lista->prox;
11  free(no);
12  return lista;
13 }
```



Tempo: $\mathrm{O}(n)$

Removendo de lista circular

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   if (no->prox == no) {
4     free(no);
5     return NULL;
6   }
7   for(ant = no->prox; ant->prox != no; ant = ant->prox);
8   ant->prox = no->prox;
9   if (lista == no)
10    lista = lista->prox;
11  free(no);
12  return lista;
13 }
```



Tempo: $O(n)$

- Podemos melhorar se soubermos o nó anterior...

Percorrendo uma lista circular

```
1 void imprimir_lista_circular(p_no lista) {
2   p_no p;
3   p = lista;
4   do {
5     printf("%d\n", p->dado);
6     p = p->prox;
7   } while (p != lista);
8 }
```

Percorrendo uma lista circular

```
1 void imprimir_lista_circular(p_no lista) {
2   p_no p;
3   p = lista;
4   do {
5     printf("%d\n", p->dado);
6     p = p->prox;
7   } while (p != lista);
8 }
```

- E se tivéssemos usado `while` ao invés de `do ... while`?

Percorrendo uma lista circular

```
1 void imprimir_lista_circular(p_no lista) {
2   p_no p;
3   p = lista;
4   do {
5     printf("%d\n", p->dado);
6     p = p->prox;
7   } while (p != lista);
8 }
```

- E se tivéssemos usado `while` ao invés de `do ... while`?
- Essa função pode ser usada com lista vazia?

Percorrendo uma lista circular

```
1 void imprimir_lista_circular(p_no lista) {
2   p_no p;
3   p = lista;
4   do {
5     printf("%d\n", p->dado);
6     p = p->prox;
7   } while (p != lista);
8 }
```

- E se tivéssemos usado `while` ao invés de `do ... while`?
- Essa função pode ser usada com lista vazia?
  - Como corrigir isso?

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M + 1)$-ésima pessoa

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M + 1)$-ésima pessoa
- Continuamos da próxima pessoa

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

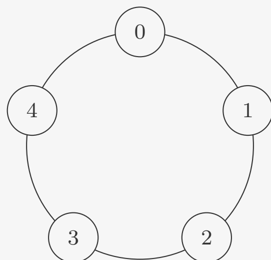
- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M + 1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

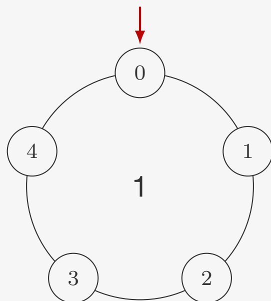


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

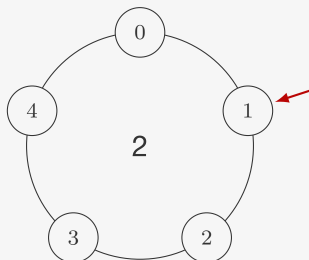


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

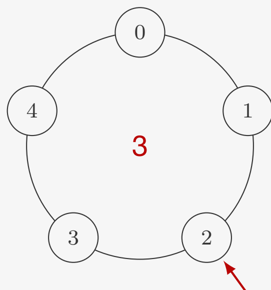


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

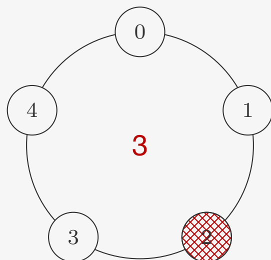


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

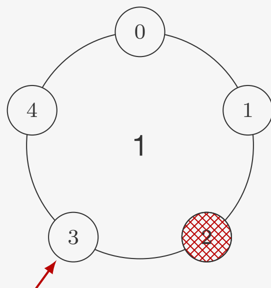


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

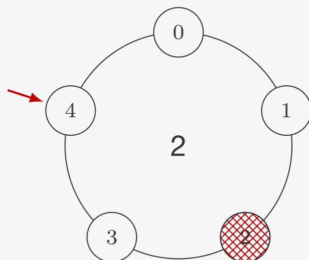


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

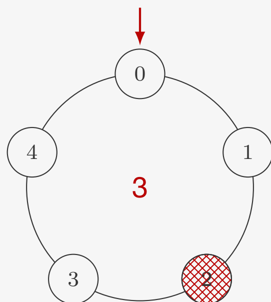


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

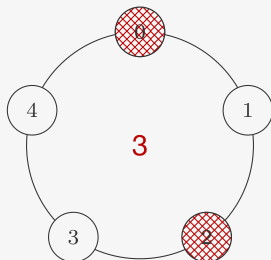


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

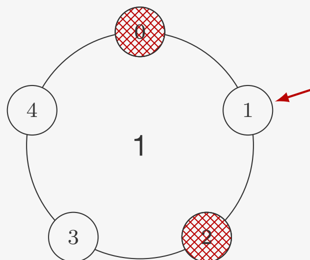


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

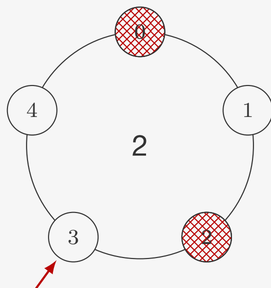


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

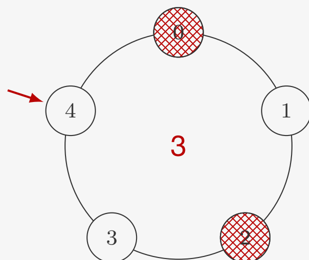


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M + 1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

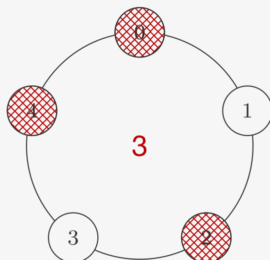


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

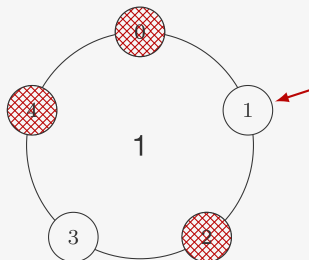


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M + 1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

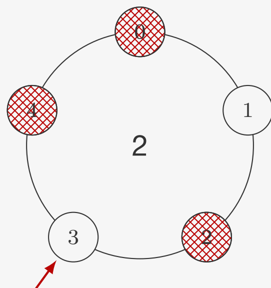


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

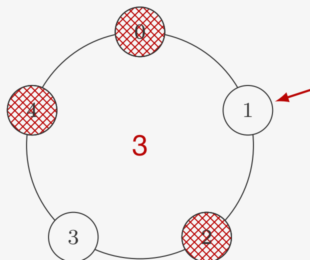


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$

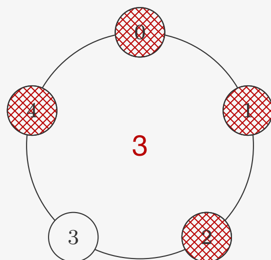


Exercício - Problema de Josephus

Vamos eleger um líder entre N pessoas

- Começamos a contar da primeira pessoa
- Contamos M pessoas
- Eliminamos $(M+1)$-ésima pessoa
- Continuamos da próxima pessoa
- Ciclamos ao chegar ao final

Exemplo: $N = 5$ e $M = 2$



Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
```

Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
```

Problema de Josephus

```c
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
```

Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
5   for (i = 0; i < N; i++)
6     lista = inserir_circular(lista, i);
```

Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
5   for (i = 0; i < N; i++)
6     lista = inserir_circular(lista, i);
7   while (lista != lista->prox) {
```

Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
5   for (i = 0; i < N; i++)
6     lista = inserir_circular(lista, i);
7   while (lista != lista->prox) {
8     for (i = 0; i < M; i++)
9       lista = lista->prox;
```

Problema de Josephus

```
1  int main() {
2    p_no lista, temp;
3    int i, N = 5, M = 2;
4    lista = criar_lista_circular();
5    for (i = 0; i < N; i++)
6      lista = inserir_circular(lista, i);
7    while (lista != lista->prox) {
8      for (i = 0; i < M; i++)
9        lista = lista->prox;
10     temp = lista->prox;
```

Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
5   for (i = 0; i < N; i++)
6     lista = inserir_circular(lista, i);
7   while (lista != lista->prox) {
8     for (i = 0; i < M; i++)
9       lista = lista->prox;
10    temp = lista->prox;
11    lista->prox = lista->prox->prox;
```

Problema de Josephus

```
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
5   for (i = 0; i < N; i++)
6     lista = inserir_circular(lista, i);
7   while (lista != lista->prox) {
8     for (i = 0; i < M; i++)
9       lista = lista->prox;
10    temp = lista->prox;
11    lista->prox = lista->prox->prox;
12    free(temp);
```

Problema de Josephus

```c
1 int main() {
2   p_no lista, temp;
3   int i, N = 5, M = 2;
4   lista = criar_lista_circular();
5   for (i = 0; i < N; i++)
6     lista = inserir_circular(lista, i);
7   while (lista != lista->prox) {
8     for (i = 0; i < M; i++)
9       lista = lista->prox;
10    temp = lista->prox;
11    lista->prox = lista->prox->prox;
12    free(temp);
13  }
14  printf("%d\n", lista->dado);
15  return 0;
16 }
```

Revistando a Inserção

O código para inserir em uma lista circular não está bom

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Precisa lidar com dois casos

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Precisa lidar com dois casos

- lista vazia ou não vazia

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Precisa lidar com dois casos

- lista vazia ou não vazia
- A remoção sofre com o mesmo problema

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Precisa lidar com dois casos

- lista vazia ou não vazia
- A remoção sofre com o mesmo problema

Tem um comportamento estranho

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Precisa lidar com dois casos

- lista vazia ou não vazia
- A remoção sofre com o mesmo problema

Tem um comportamento estranho

- Inserimos após o primeiro elemento

Revistando a Inserção

O código para inserir em uma lista circular não está bom

```
1
2 p_no inserir_circular(p_no lista, int x) {
3   p_no novo;
4   novo = malloc(sizeof(No));
5   novo->dado = x;
6   if (lista == NULL)
7     novo->prox = novo;
8   else {
9     novo->prox = lista->prox;
10    lista->prox = novo;
11  }
12  return novo;
```

Precisa lidar com dois casos

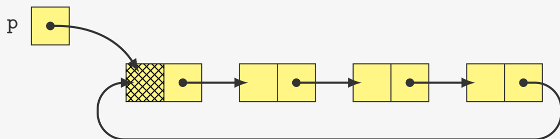
- lista vazia ou não vazia
- A remoção sofre com o mesmo problema

Tem um comportamento estranho

- Inserimos após o primeiro elemento
- Mudamos quem é o primeiro elemento da lista

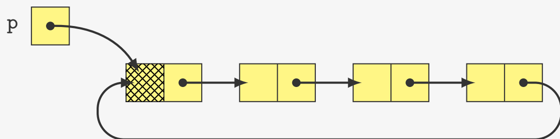
Listas circulares com cabeça

Lista circular com cabeça:

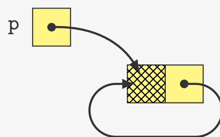


Listas circulares com cabeça

Lista circular com cabeça:

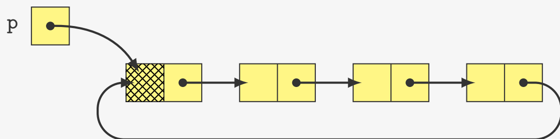


Lista circular vazia:

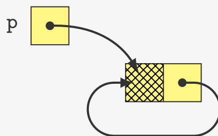


Listas circulares com cabeça

Lista circular com cabeça:



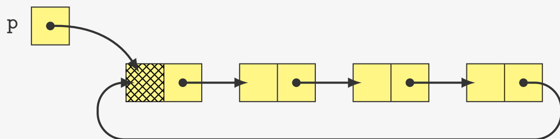
Lista circular vazia:



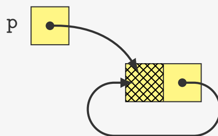
Diferenças para a versão sem cabeça:

Listas circulares com cabeça

Lista circular com cabeça:



Lista circular vazia:

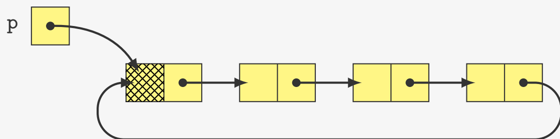


Diferenças para a versão sem cabeça:

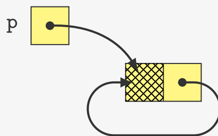
- lista sempre aponta para o nó *dummy*

Listas circulares com cabeça

Lista circular com cabeça:



Lista circular vazia:

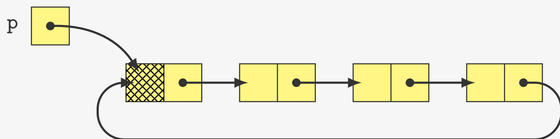


Diferenças para a versão sem cabeça:

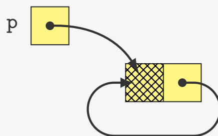
- lista sempre aponta para o nó *dummy*
- código de inserção e de remoção mais simples

Listas circulares com cabeça

Lista circular com cabeça:



Lista circular vazia:



Diferenças para a versão sem cabeça:

- lista sempre aponta para o nó *dummy*
- código de inserção e de remoção mais simples
- ao percorrer tem que ignorar cabeça

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

```
1 p_no remover_circular(p_no lista, p_no no) {
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
```

Inserção e remoção simplificadas

```c
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

```c
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   for(ant = no->prox; ant->prox != no; ant = ant->prox);
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   for(ant = no->prox; ant->prox != no; ant = ant->prox);
4   ant->prox = no->prox;
```

Inserção e remoção simplificadas

```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

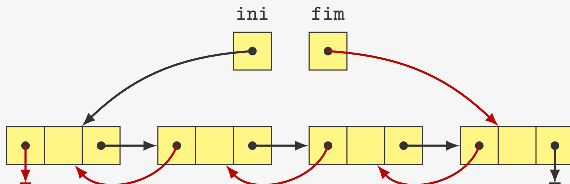
```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   for(ant = no->prox; ant->prox != no; ant = ant->prox);
4   ant->prox = no->prox;
5   free(no);
```

Inserção e remoção simplificadas

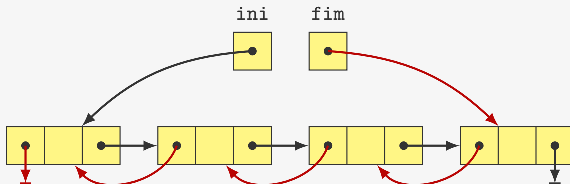
```
1 p_no inserir_circular(p_no lista, int x) {
2   p_no novo;
3   novo = malloc(sizeof(No));
4   novo->dado = x;
5   novo->prox = lista->prox;
6   lista->prox = novo;
7   return lista;
8 }
```

```
1 p_no remover_circular(p_no lista, p_no no) {
2   p_no ant;
3   for(ant = no->prox; ant->prox != no; ant = ant->prox);
4   ant->prox = no->prox;
5   free(no);
6   return lista;
7 }
```

Variações - Duplamente ligada



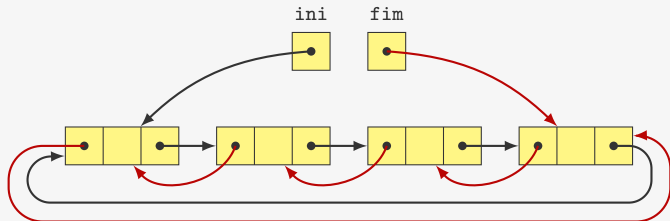
Variações - Duplamente ligada



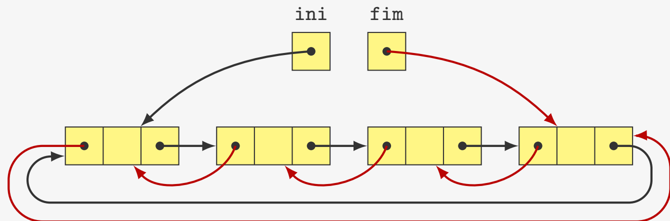
Exemplos:

- Operações desfazer/refazer em software
- Player de música (música anterior e próxima música)

Variações - Lista dupla circular

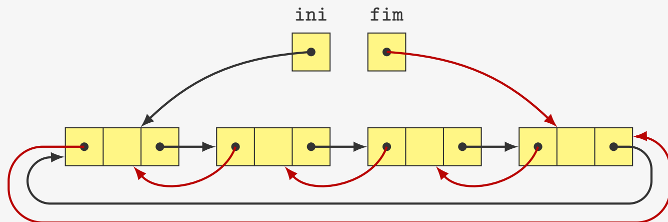


Variações - Lista dupla circular



Permite inserção e remoção em $O(1)$

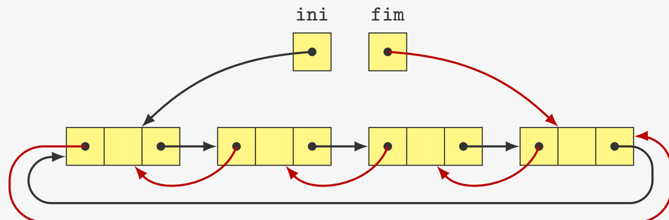
Variações - Lista dupla circular



Permite inserção e remoção em $O(1)$

- Variável `fim` é opcional (`fim == ini->ant`)

Variações - Lista dupla circular



Permite inserção e remoção em $O(1)$

- Variável `fim` é opcional (`fim == ini->ant`)

Podemos ter uma lista dupla circular com cabeça também...

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`
- Poderíamos esperar para liberar nós no final do programa

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`
- Poderíamos esperar para liberar nós no final do programa
  - Podemos reutilizar esses nós

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`
- Poderíamos esperar para liberar nós no final do programa
  - Podemos reutilizar esses nós
  - Diminuímos as chamadas para `free` e `malloc`

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`
- Poderíamos esperar para liberar nós no final do programa
  - Podemos reutilizar esses nós
  - Diminuímos as chamadas para `free` e `malloc`

Como fazer esse controle?

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`
- Poderíamos esperar para liberar nós no final do programa
  - Podemos reutilizar esses nós
  - Diminuímos as chamadas para `free` e `malloc`

Como fazer esse controle?

- Podemos usar uma lista ligada dos nós não utilizados

Gerenciamento de memória

No nosso código para listas ligadas, toda vez que:

- Precisamos de um novo nó, fazemos um `malloc`
- Toda vez que removemos um nó, fazemos um `free`

Alternativa:

- Poderíamos alocar muitos nós de uma vez só
  - fazendo poucas chamadas para `malloc`
- Poderíamos esperar para liberar nós no final do programa
  - Podemos reutilizar esses nós
  - Diminuímos as chamadas para `free` e `malloc`

Como fazer esse controle?

- Podemos usar uma lista ligada dos nós não utilizados
- É o que chamamos de lista livre

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez
  - Colocamos esses nós na lista livre

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez
  - Colocamos esses nós na lista livre
- Quando precisamos de um nó, pegamos da lista livre

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez
  - Colocamos esses nós na lista livre
- Quando precisamos de um nó, pegamos da lista livre
  - Se faltar nós, alocamos mais um grande número de nós

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez
  - Colocamos esses nós na lista livre
- Quando precisamos de um nó, pegamos da lista livre
  - Se faltar nós, alocamos mais um grande número de nós
- Quando liberamos um nó, inserimos ele na lista livre

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez
  - Colocamos esses nós na lista livre
- Quando precisamos de um nó, pegamos da lista livre
  - Se faltar nós, alocamos mais um grande número de nós
- Quando liberamos um nó, inserimos ele na lista livre

A lista livre pode armazenar os nós não utilizados de todo o programa, mesmo que tenhamos várias listas

Lista Livre

Ideia:

- Alocamos um grande número de nós de uma só vez
  - Colocamos esses nós na lista livre
- Quando precisamos de um nó, pegamos da lista livre
  - Se faltar nós, alocamos mais um grande número de nós
- Quando liberamos um nó, inserimos ele na lista livre

A lista livre pode armazenar os nós não utilizados de todo o programa, mesmo que tenhamos várias listas

- não é necessário ter uma lista livre para cada lista ligada

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
5     lista_livre = malloc(N * sizeof(No));
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
5     lista_livre = malloc(N * sizeof(No));
6     for (i = 0; i < N - 1; i++)
7       lista_livre[i].prox = &lista_livre[i+1];
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
5     lista_livre = malloc(N * sizeof(No));
6     for (i = 0; i < N - 1; i++)
7       lista_livre[i].prox = &lista_livre[i+1];
8     lista_livre[N - 1].prox = NULL;
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
5     lista_livre = malloc(N * sizeof(No));
6     for (i = 0; i < N - 1; i++)
7       lista_livre[i].prox = &lista_livre[i+1];
8     lista_livre[N - 1].prox = NULL;
9   }
10  primeiro = lista_livre;
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
5     lista_livre = malloc(N * sizeof(No));
6     for (i = 0; i < N - 1; i++)
7       lista_livre[i].prox = &lista_livre[i+1];
8     lista_livre[N - 1].prox = NULL;
9   }
10  primeiro = lista_livre;
11  lista_livre = lista_livre->prox;
```

Lista Livre - Alocando um nó

```c
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```c
1 p_no novo_no() {
2   int i;
3   p_no primeiro;
4   if (lista_livre == NULL) {
5     lista_livre = malloc(N * sizeof(No));
6     for (i = 0; i < N - 1; i++)
7       lista_livre[i].prox = &lista_livre[i+1];
8     lista_livre[N - 1].prox = NULL;
9   }
10  primeiro = lista_livre;
11  lista_livre = lista_livre->prox;
12  return primeiro;
13 }
```

Lista Livre - Alocando um nó

```
1 #define N 1000
2
3 p_no lista_livre = NULL;
```

```
1  p_no novo_no() {
2    int i;
3    p_no primeiro;
4    if (lista_livre == NULL) {
5      lista_livre = malloc(N * sizeof(No));
6      for (i = 0; i < N - 1; i++)
7        lista_livre[i].prox = &lista_livre[i+1];
8      lista_livre[N - 1].prox = NULL;
9    }
10   primeiro = lista_livre;
11   lista_livre = lista_livre->prox;
12   return primeiro;
13 }
```

```
1 p_no adicionar_elemento(p_no lista, int x) {
2   p_no novo;
3   novo = novo_no();
4   novo->dado = x;
5   novo->prox = lista;
6   return novo;
7 }
```

Lista Livre - Liberando um nó

```
1 void libera_no(p_no no) {
2   no->prox = lista_livre;
3   lista_livre = no;
4 }
```

Lista Livre - Liberando um nó

```c
1 void libera_no(p_no no) {
2   no->prox = lista_livre;
3   lista_livre = no;
4 }
```

```c
1 void destruir_lista_rec(p_no lista) {
2   if(lista != NULL) {
3     destruir_lista_rec(lista->prox);
4     libera_no(lista);
5   }
6 }
```

Lista Livre

Outra opção é:

Lista Livre

Outra opção é:

- se soubermos o número máximo de nós a serem utilizados

Lista Livre

Outra opção é:

- se soubermos o número máximo de nós a serem utilizados
- podemos alocá-los todos de uma vez só

Lista Livre

Outra opção é:

- se soubermos o número máximo de nós a serem utilizados
- podemos alocá-los todos de uma vez só

A implementação apresentada tem um problema:

Lista Livre

Outra opção é:

- se soubermos o número máximo de nós a serem utilizados
- podemos alocá-los todos de uma vez só

A implementação apresentada tem um problema:

- Não liberamos os blocos de nós alocados

Lista Livre

Outra opção é:

- se soubermos o número máximo de nós a serem utilizados
- podemos alocá-los todos de uma vez só

A implementação apresentada tem um problema:

- Não liberamos os blocos de nós alocados
- Precisamos dos endereços dos blocos para isso

Lista Livre

Outra opção é:

- se soubermos o número máximo de nós a serem utilizados
- podemos alocá-los todos de uma vez só

A implementação apresentada tem um problema:

- Não liberamos os blocos de nós alocados
- Precisamos dos endereços dos blocos para isso
  - Poderíamos ter uma outra lista com esses ponteiros

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

- precisa lidar com blocos de tamanhos diferentes

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

- precisa lidar com blocos de tamanhos diferentes

Muitas linguagens também fazem `free` automático

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

- precisa lidar com blocos de tamanhos diferentes

Muitas linguagens também fazem `free` automático

- *coleta de lixo*

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

- precisa lidar com blocos de tamanhos diferentes

Muitas linguagens também fazem `free` automático

- *coleta de lixo*
- Exemplo: contagem de referências

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

- precisa lidar com blocos de tamanhos diferentes

Muitas linguagens também fazem `free` automático

- *coleta de lixo*
- Exemplo: contagem de referências
  - quantos ponteiros apontam para a célula?

Alocação dinâmica de memória

O `malloc` faz algo parecido com lista livre, mas é diferente

- precisa lidar com blocos de tamanhos diferentes

Muitas linguagens também fazem `free` automático

- *coleta de lixo*
- Exemplo: contagem de referências
  - quantos ponteiros apontam para a célula?
  - se for zero, podemos liberar a célula

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1 typedef union elemento {
2   int inteiro;
3   float real;
4 } elemento;
```

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1 typedef union elemento {
2   int inteiro;
3   float real;
4 } elemento;
```

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1  typedef union elemento {
2    int inteiro;
3    float real;
4  } elemento;
5
6  int main() {
7    elemento x;
8    x.inteiro = 1;
9    printf("%d, %f\n", x.inteiro, x.real);
10   x.real = 1;
11   printf("%d, %f\n", x.inteiro, x.real);
12 }
```

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1  typedef union elemento {
2    int inteiro;
3    float real;
4  } elemento;
5
6  int main() {
7    elemento x;
8    x.inteiro = 1;
9    printf("%d, %f\n", x.inteiro, x.real);
10   x.real = 1;
11   printf("%d, %f\n", x.inteiro, x.real);
12 }
```

O que é impresso?

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1  typedef union elemento {
2    int inteiro;
3    float real;
4  } elemento;
5
6  int main() {
7    elemento x;
8    x.inteiro = 1;
9    printf("%d, %f\n", x.inteiro, x.real);
10   x.real = 1;
11   printf("%d, %f\n", x.inteiro, x.real);
12 }
```

O que é impresso?

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1  typedef union elemento {
2    int inteiro;
3    float real;
4  } elemento;
5
6  int main() {
7    elemento x;
8    x.inteiro = 1;
9    printf("%d, %f\n", x.inteiro, x.real);
10   x.real = 1;
11   printf("%d, %f\n", x.inteiro, x.real);
12 }
```

O que é impresso?

1, 0.000000

Listas Generalizadas

Uma lista ligada pode ser definida de forma recursiva:

- ou é uma lista vazia
- ou é composta de um elemento seguido de uma lista

Usualmente, o elemento é de um tipo determinado (e.g. `int`)

Usando `union`, podemos ter elementos de vários tipos

- Uma sequência de bits é interpretada de várias formas
- Ex: posso considerar como um `int` ou como um `float`

```
1  typedef union elemento {
2    int inteiro;
3    float real;
4  } elemento;
5
6  int main() {
7    elemento x;
8    x.inteiro = 1;
9    printf("%d, %f\n", x.inteiro, x.real);
10   x.real = 1;
11   printf("%d, %f\n", x.inteiro, x.real);
12 }
```

O que é impresso?

1, 0.000000

1065353216, 1.000000

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista
- `((), ((1), (2, 3)), 4)` - uma lista com

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista
- `((), ((1), (2, 3)), 4)` - uma lista com
  - o primeiro elemento `()` é uma lista vazia

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista
- `((), ((1), (2, 3)), 4)` - uma lista com
  - o primeiro elemento `()` é uma lista vazia
  - o segundo elemento `((1), (2,3))` é uma lista com

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista
- `((), ((1), (2, 3)), 4)` - uma lista com
  - o primeiro elemento `()` é uma lista vazia
  - o segundo elemento `((1), (2,3))` é uma lista com
    - uma lista `(1)` de um elemento atômico

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista
- `((), ((1), (2, 3)), 4)` - uma lista com
  - o primeiro elemento `()` é uma lista vazia
  - o segundo elemento `((1), (2,3))` é uma lista com
    - uma lista `(1)` de um elemento atômico
    - uma lista `(2, 3)` de dois elementos atômicos

Listas Generalizadas

Nas listas generalizadas, os elementos da lista podem ser:

- átomos (e.g. `int`, `float`, `char *`)
- ou outras listas generalizadas

É a estrutura de dados básica de linguagens como o LISP

- LISP: LISt Processor

Exemplos:

- `()` - lista vazia
- `(1, 2, 3, 4)` - átomos de mesmo tipo
- `(1, 'A', 3, 4.5)` - átomos de tipos diferentes
- `(1, (2, 3), 4)` - o segundo elemento é uma lista
- `(1, ('A', 3), 4.5)` - o segundo elemento é uma lista
- `((), ((1), (2, 3)), 4)` - uma lista com
  - o primeiro elemento `()` é uma lista vazia
  - o segundo elemento `((1), (2,3))` é uma lista com
    - uma lista `(1)` de um elemento atômico
    - uma lista `(2, 3)` de dois elementos atômicos
  - o terceiro elemento `4` é um átomo

Listas Generalizadas - Implementação

```
1 enum elemento_t {int_t, float_t, char_t, lista_t};
2
3 typedef union elemento {
4   int inteiro;
5   float real;
6   char caracter;
7   struct No* lista;
8 } elemento;
```

Listas Generalizadas - Implementação

```
1  enum elemento_t {int_t, float_t, char_t, lista_t};
2
3  typedef union elemento {
4    int inteiro;
5    float real;
6    char caracter;
7    struct No* lista;
8  } elemento;
9
10 typedef struct No {
11   enum elemento_t tipo;
12   elemento valor;
13   struct No* prox;
14 } No;
15
16 typedef No * p_no;
```

Listas Generalizadas - Implementação

```
1 void imprime_rec(p_no lista) {
2   if(lista == NULL)
3     return;
```

Listas Generalizadas - Implementação

```
1 void imprime_rec(p_no lista) {
2   if(lista == NULL)
3     return;
4   if(lista->tipo == lista_t) {
5     printf("(");
6     imprime_rec(lista->valor.lista);
7     printf(")");
8   }
```

Listas Generalizadas - Implementação

```
1 void imprime_rec(p_no lista) {
2   if(lista == NULL)
3     return;
4   if(lista->tipo == lista_t) {
5     printf("(");
6     imprime_rec(lista->valor.lista);
7     printf(")");
8   }
9   else if(lista->tipo == int_t)
10    printf("%d", lista->valor.inteiro);
```

Listas Generalizadas - Implementação

```
1  void imprime_rec(p_no lista) {
2    if(lista == NULL)
3      return;
4    if(lista->tipo == lista_t) {
5      printf("(");
6      imprime_rec(lista->valor.lista);
7      printf(")");
8    }
9    else if(lista->tipo == int_t)
10     printf("%d", lista->valor.inteiro);
11   else if(lista->tipo == float_t)
12     printf("%f", lista->valor.real);
```

Listas Generalizadas - Implementação

```
1 void imprime_rec(p_no lista) {
2   if(lista == NULL)
3     return;
4   if(lista->tipo == lista_t) {
5     printf("(");
6     imprime_rec(lista->valor.lista);
7     printf(")");
8   }
9   else if(lista->tipo == int_t)
10    printf("%d", lista->valor.inteiro);
11  else if(lista->tipo == float_t)
12    printf("%f", lista->valor.real);
13  else if(lista->tipo == char_t)
14    printf("%c", lista->valor.caracter);
```

Listas Generalizadas - Implementação

```
1  void imprime_rec(p_no lista) {
2    if(lista == NULL)
3      return;
4    if(lista->tipo == lista_t) {
5      printf("(");
6      imprime_rec(lista->valor.lista);
7      printf(")");
8    }
9    else if(lista->tipo == int_t)
10     printf("%d", lista->valor.inteiro);
11   else if(lista->tipo == float_t)
12     printf("%f", lista->valor.real);
13   else if(lista->tipo == char_t)
14     printf("%c", lista->valor.caracter);
15   if(lista->prox != NULL)
16     printf(", ");
17   imprime_rec(lista->prox);
18 }
```

Listas Generalizadas - Implementação

```
1 void imprime_rec(p_no lista) {
2   if(lista == NULL)
3     return;
4   if(lista->tipo == lista_t) {
5     printf("(");
6     imprime_rec(lista->valor.lista);
7     printf(")");
8   }
9   else if(lista->tipo == int_t)
10    printf("%d", lista->valor.inteiro);
11  else if(lista->tipo == float_t)
12    printf("%f", lista->valor.real);
13  else if(lista->tipo == char_t)
14    printf("%c", lista->valor.caracter);
15  if(lista->prox != NULL)
16    printf(", ");
17  imprime_rec(lista->prox);
18 }
19
20 void imprime(p_no lista) {
21   printf("(");
22   imprime_rec(lista);
23   printf(")\n");
24 }
```

Exercício

Representente polinômios utilizando listas ligadas e apresente uma função que soma dois polinômios.

Exercício

Implemente a operação *inserir elemento* de uma lista duplamente ligada.

Exercício

Escreva uma função que devolve a concatenação de duas listas circulares dadas. Sua função pode destruir a estrutura das listas dadas.

Exercício

Escreva uma função que dada uma lista duplamente ligada (sem cabeça) e dois de seus nós, troca os dois nós de lugar na lista.

Exercício

Faça uma função que insere um elemento (atômico ou não) em uma lista generalizada.