Estruturas de Dados - Lista Encadeada

Aula passada

O que vimos:

- Introdução a estruturas de dados
- Lista
- Tipos de alocação
  - sequencial
  - encadeada
- Lista sequencial
- Introdução a lista encadeada

Lista encadeada

- Em uma lista encadeada, alocamos memória dinamicamente, ou seja, quando há necessidade
  - evita um eventual desperdício de memória (evita caso sonde alocamos mais espaço do que utilizaremos)
- Portanto, já sabemos que a lista encadeada é mais eficiente do ponto de vista de memória
- Pergunta: o uso da lista encadeada evita todo problema de memória?
- Em outras palavras: ainda podemos ter problema de memória caso utilizemos a lista encadeada?

Lista encadeada

- Lista formada por uma sequência de nós
- Nós estão dispostos em posições arbitrárias de memória
- A ordem dos elementos na lista é estabelecida pelo uso de nós para o armazenamento dos elementos
- Cada nó v é formado por dois campos:
  - $v \rightarrow chave$: guarda o elemento
  - $v \rightarrow prox$: indica a localização do nó que o sucede na lista
- Podemos considerar que $v \rightarrow prox$ ao mesmo tempo representa o nó seguinte a v, e representa a ideia de uma lista de nós após v

Lista encadeada

- Para representar a ausência de nó, utilizaremos um nó especial λ
  - λ ao mesmo tempo representa a ausência de nó, e representa a ideia de uma lista nula
  - se $v = \lambda$, então v é um nó nulo
  - se $v \rightarrow prox = \lambda$, então v não tem um nó seguinte a ele
- Trabalharemos com a ideia de um nó vazio como nó inicial da lista
  - chamaremos este nó de nó cabeça da lista
  - o nó cabeça não pertence de fato à lista; ele só existe para facilitar nosso trabalho!
  - o "corpo" da lista é a sequência de nós seguintes ao nó cabeça
- Para passarmos uma lista encadeada como parâmetro, basta passarmos o nó cabeça

Lista encadeada: criar

Algoritmo: CriarListaEncadeada()

Saída: no inicial vazio da lista encadeada

1 criar novo nó v

2 $v \rightarrow prox = \lambda$

3 **retorne** v

Complexidade: $O(1)$

Lista encadeada: buscar

Algoritmo: BuscarEmListaEncadeada(v, x)

Entrada: nó inicial v, valor x

Saída: nó cuja chave é x, ou λ se não há nó com chave x na lista

```
1 enquanto v→prox ≠ λ faça
2 |   v = v→prox
3 |   se v→chave == x então
4 |   |   retorne v
5 retorne λ
```

Complexidade: $O(n)$

Lista encadeada: incluir

Algoritmo: IncluirEmListaEncadeada(v, x)

Entrada: nó inicial v, valor x

```
1 criar novo nó u
2 u→chave = x
3 u→prox = λ
4 enquanto v→prox ≠ λ faça
5 |   v = v→prox
6 v→prox = u
```

Complexidade: $O(n)$

Lista encadeada: incluir

Algoritmo: IncluirEmListaEncadeada(v, x)

Entrada: nó inicial v, valor x

```
1 criar novo nó u
2 u→chave = x
3 u→prox = λ
4 enquanto v→prox ≠ λ faça
5 |   v = v→prox
6 v→prox = u
```

Complexidade: $O(n)$

Pergunta: este método é eficiente?

Lista encadeada: incluir

- Em uma lista, não há necessidade de mantermos a ordem de inserção dos elementos
  - o importante é garantir que um elemento inserido continue na lista até ser removido (ou seja, não haja perca de dados)
- Em uma lista encadeada, a posição onde inserimos o novo nó influencia na complexidade
- Pergunta: podemos escrever um método de inserção mais eficiente que o anterior (cuja complexidade é $O(n)$)?
- Em outras palavras: onde devemos adicionar o novo nó para obtermos uma complexidade menor?

Lista encadeada: incluir

Algoritmo: IncluirEmListaEncadeada(v, x)

Entrada: nó inicial v, valor x

1 criar novo nó u
2 $u \rightarrow chave = x$
3 $u \rightarrow prox = v \rightarrow prox$
4 $v \rightarrow prox = u$

Complexidade: $O(1)$

Lista encadeada: excluir

Algoritmo: ExcluirEmListaEncadeada(v, x)

Entrada: nó inicial v, valor x

Saída: nó excluído, ou λ se x não pertence à lista

```
1 enquanto v→prox ≠ λ e v→prox→chave ≠ x faça
2 |   v = v→prox
3 se v→prox == λ então
4 |   retorne λ
5 r = v→prox
6 v→prox = r→prox
7 retorne r
```

Complexidade: $O(n)$

Lista circular

- Lista duplamente encadeada
- Cada nó guarda informações sobre dois nós: antecessor e sucessor
- Em listas circulares, um nó v é uma estrutura com as seguintes informações:
  - $v \rightarrow chave$: guarda o elemento
  - $v \rightarrow prox$: indica a localização do nó que o sucede na lista
  - $v \rightarrow ant$: indica a localização do nó que o antecede na lista
- Para passarmos uma lista circular como parâmetro, basta passarmos o nó cabeça da lista
  - novamente, utilizaremos um nó vazio como nó cabeça
- O nó anterior ao nó cabeça é o último, e o nó sucessor do nó cabeça é o primeiro
- Não há necessidade de sentinela para verificarmos o fim da lista
- Métodos ficam mais simples se comparado à lista encadeada

Lista circular: criar

Algoritmo: CriarListaCircular()

Saída: no inicial vazio da lista circular

1 criar novo nó v

2 $v \rightarrow prox = v$

3 $v \rightarrow ant = v$

4 **retorne** v

Complexidade: $O(1)$

Lista circular: buscar

Algoritmo: BuscarEmListaCircular(v, x)

Entrada: nó inicial v, valor x

Saída: nó cuja chave é x, ou λ se não há nó com chave x na lista

```
1 u = v
2 enquanto v→prox ≠ u faça
3 |   v = v→prox
4 |   se v→chave == x então
5 |   |   retorne v
6 retorne λ
```

Complexidade: $O(n)$

Lista circular: incluir (no início da lista)

Algoritmo: IncluirEmListaCircular(v, x)

Entrada: nó inicial v, valor x

1 criar novo nó u
2 $u \rightarrow chave = x$
3 $u \rightarrow prox = v \rightarrow prox$
4 $u \rightarrow ant = v$
5 $v \rightarrow prox = u$
6 $u \rightarrow prox \rightarrow ant = u$

Complexidade: $O(1)$

Lista circular: incluir (no final da lista)

Algoritmo: IncluirEmListaCircular(v, x)

Entrada: nó inicial v, valor x

1 criar novo nó u
2 $u \rightarrow chave = x$
3 $u \rightarrow prox = v$
4 $u \rightarrow ant = v \rightarrow ant$
5 $v \rightarrow ant \rightarrow prox = u$
6 $v \rightarrow ant = u$

Complexidade: $O(1)$

Lista circular: excluir

Algoritmo: ExcluirEmListaCircular(v, x)

Entrada: nó inicial v, valor x

Saída: nó excluído, ou λ se não há nó com chave x na lista

```
1 r = BuscarEmListaCircular(v, x)
2 se r ≠ λ então
3 |   r→ant→prox = r→prox
4 |   r→prox→ant = r→ant
5 retorne r
```

Complexidade: $O(n)$