Listas Lineares Sequenciais

Algoritmos e Estruturas de Dados

Prof. Dr. Luciano Demétrio Santos Pacífico

{luciano.pacifico@ufrpe.br}

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Conteúdo

- **Introdução**
- **Alocação Sequencial**
- **Pilhas e Filas**

Introdução

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Listas Lineares

- Dentre as estruturas de dados não primitivas, as listas lineares são as de manipulação mais simples.

- Uma lista linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, estão relacionados entre si.

  - Ex.: Lista de funcionários de uma empresa, notas de compras, notas de alunos de uma disciplina, ...

Listas Lineares

- Uma lista linear ou tabela é um conjunto de $n \geq 0$ elementos (*nós*) $L[1], L[2], ..., L[n]$, tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos mesmos dentro de uma sequência linear.

  - Se $n > 0$, $L[1]$ é o primeiro nó;
  - Para $1 < k \leq n$, o nó $L[k]$ é precedido pelo nó $L[k - 1]$.

Listas Lineares

- As operações mais frequentes em listas lineares, assim como na maioria das estruturas de dados, são a **busca**, a **inserção** de novos elementos e a **remoção** de elementos.

- Essas operações são consideradas básicas, sendo necessário que os algoritmos que as implementam sejam eficientes.

- Outras operações: edição de um elemento, concatenação entre duas listas, ordenação dos elementos, cardinalidade, etc.

Listas Lineares – Casos Particulares

- ***Deques*** (*double ended queue*): remoções e inserções permitidas apenas nas extremidades da lista.

- **Pilhas**: remoções e inserções permitidas apenas em uma única extremidade da fila.

- **Fila**: inserções realizadas em um extremo, remoções no outro.

Alocação Sequencial

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Alocação Sequencial e Encadeada

- O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa (sempre contígua ou não) na memória de dois nós consecutivos na lista.

- Quando os elementos estão armazenados em posições contíguas da memória, temos a **alocação sequencial**.

- Caso contrário, temos a **alocação encadeada**.

Alocação Sequencial

- A maneira mais simples de se manter uma lista na memória do computador é colocar seus nós em posições contíguas.

- O endereço real do (j+1)-ésimo elemento da lista encontra-se c unidades adiante daquele correspondente ao j-ésimo elemento, onde c é o número de palavras de memória que cada nó ocupa.

Alocação Sequencial

- Como a implementação da alocação sequencial em linguagens de alto nível é geralmente realizada com a reserva prévia de memória para cada estrutura (**alocação estática**), a inserção e remoção de nós não ocorre de fato.

- Em vez disso, usa-se uma "simulação" dessas operações, geralmente através de variáveis que indiquem os limites da memória realmente utilizados.

Alocação Sequencial

- O armazenamento sequencial é muito atraente para o caso de pilhas e filas, porque nessas estruturas as operações básicas podem ser implementadas de forma bastante eficiente.

- Porém, ao fazer uso da alocação estática, deve-se ter em mente que a estrutura vai limitar-se ao armazenamento de no máximo *M* nós.

Operações em Listas Genéricas

- Seja uma lista linear *L* qualquer.
- Cada nó de *L* é formado por campos, que armazenam as características dos elementos de *L*.
- Cada nó de *L* possui um identificador distinto, geralmente denotado **chave**.
- Os nós podem encontrar-se ordenados ou não, de acordo com suas chaves.
- Caso os nós estejam ordenados, dizemos que *L* é **lista ordenada**; caso contrário, *L* é **lista não ordenada**.

Estruturas Básicas

- Para facilitar a organização dos códigos para manipulação de Listas Lineares Sequenciais, definiremos duas Estruturas de Dados básicas:
  - O Nó de uma Lista Linear Sequencial, que conterá um objeto da aplicação, que, para nosso exemplo, deve conter unicamente o atributo **chave**;

```
1. registro NoListaSeq
2.     chave:inteiro
```

  - A Lista Linear Sequencial propriamente dita, que consistirá de um vetor (**Array**) com capacidade máxima igual a ***M*** nós.

```
1. registro ListaSeq
2.     maximo:inteiro, //Número máximo de objetos
3.     ultimo:inteiro, //Última posição alocada
4.     dados:Array<NoListaSeq>
```

Busca de Elemento

- Assumindo que nossa pseudolinguagem ofereça no máximo apenas um tipo retorno, a busca de uma chave ***x*** em uma lista linear sequencial pode ser realizada através dos procedimentos abaixo:

```
1. //x -> chave do nó procurado, lista -> ListaSeq
2. procedimento buscar(x, lista)
3.     indice = buscarIndice(x, lista)
4.     se indice > 0
5.         retorne lista.dados[indice]
6.     senão
7.         retorne NIL //NIL -> valor Nulo
```

Busca de Elemento

- Assumindo que nossa pseudolinguagem ofereça no máximo apenas um tipo retorno, a busca de uma chave ***x*** em uma lista linear sequencial pode ser realizada através dos procedimentos abaixo: (Cont.)

```
1.  //x -> chave do nó procurado, lista -> ListaSeq
2.  procedimento buscarIndice(x, lista)
3.      indice = 0                  //o indice será 0 se x não pertence à lista
4.      n = lista.ultimo            //n indica a última posição ocupada na lista
5.      L = lista.dados             //L armazena o Array que contém os dados da lista
6.      i = 1                       //i será a variável iterador que percorrerá a lista
7.      enquanto i <= n
8.          se L[i].chave == x
9.              indice = i          //chave x encontrada na posição i
10.             i = n + 1           //força saída do laço
11.         senão
12.             i = i + 1
13.     retorne indice
```

Busca de Elemento

- O procedimento `buscarIndice` tem complexidade de pior caso na ordem de $O(n)$, dado que no pior caso a lista estará saturada (com ***n = M*** posições preenchidas), e a chave buscada ***x*** estará ou na última posição ou não estará na lista.

- De modo semelhante, o procedimento `buscar` terá complexidade de pior caso na ordem $O(n)$, tendo em vista que o mesmo faz uso do `buscarIndice`, e que suas outras operações possuem complexidade constante $O(1)$.

- Quando a lista é **ordenada**, podemos interromper a busca caso o número procurado não pertença a mesma.

Busca de Elemento

- Quando a lista é **ordenada**, podemos interromper a busca caso o número procurado não pertença a mesma.
- Não há necessidade de continuar a busca após o valor da chave ***x*** ser excedido (em listas em ordem crescente).
- O procedimento `buscar` não sobre modificações, sendo necessária apenas a substituição do procedimento `buscarIndice` pelo `buscarIndiceOrdenada`.

```
1.  //x -> chave do nó procurado, lista -> ListaSeq ordenada em ordem crescente
2.  procedimento buscarIndiceOrdenada(x, lista)
3.      indice = 0                  //o indice será 0 se x não pertence à lista
4.      n = lista.ultimo            //n indica a última posição ocupada na lista
5.      L = lista.dados             //L armazena o Array que contém os dados da lista
6.      i = 1                       //i será a variável iterador que percorrerá a lista
7.      enquanto (i <= n) e (L[i].chave < x)
8.          i = i + 1
9.      se L[i].chave == x
10.         indice = i              //x encontrado na posição i
11.     retorne indice
```

Inserção e Remoção

- Para o caso da inserção e da remoção consideraremos uma lista linear não ordenada qualquer com *M* posições.

- Problemas possíveis:

  - Tentativa de inserção em uma lista totalmente ocupada (*overflow*);

  - Tentativa de remoção em uma lista vazia (*underflow*).

Inserção

- Como o atributo chave de cada nó deverá servir como seu identificador único, no procedimento inserir apresentado abaixo, não será permitida a inserção de chaves replicadas, evitando-se ambiguidades nas operações de busca e remoção.

```
1.  //X -> NoListaSeq a ser inserido, lista -> ListaSeq
2.  procedimento inserir(X, lista)
3.      n = lista.ultimo
4.      se n < lista.maximo
5.          se buscarIndice(X.chave, lista) == 0
6.              n = n + 1
7.              lista.dados[n] = X
8.              lista.ultimo = n
9.          senão
10.             imprimir("Nó "+ X.chave + " já existe!")
11.     senão
12.         imprimir("Lista cheia!")
```

Remoção

- No código abaixo, optou-se por retornar o nó a ser removido (caso exista), pois em algumas aplicações pode desejar-se fazer uso das informações contidas nesse nó antes de sua exclusão definitiva.

```
1.  //x -> chave do nó a ser removido, lista -> ListaSeq
2.  procedimento remover(x, lista)
3.      removido = NIL
4.      n = lista.ultimo
5.      indice = buscarIndice(x, lista)
6.      L = lista.dados
7.      se indice != 0
8.          removido = L[indice]
9.          para i = indice até n - 1
10.             L[i] = L[i + 1]
11.         lista.ultimo = n - 1
12.     senão
13.         imprimir("Nó "+ x + " não existe!")
14.     retorne removido
```

Alocação Sequencial

- Tanto o procedimento *inserir* quanto o *remover* apresentam complexidade de pior caso da ordem de $O(n)$.

Pilhas e Filas

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Pilhas

- Em uma **pilha**, as **inserções** e **remoções** ocorrerão sempre **na mesma extremidade da estrutura**.

- Essa extremidade é conhecida por **topo** da pilha.

- Uma pilha **não suporta a operação de busca**.
  - Operações são realizadas apenas no topo, sendo esse o único elemento que pode ser consultado.

- O último elemento inserido em uma pilha será o primeiro elemento a ser removido (política **LIFO**, ou *last-in-first-out*).

Pilhas – Inserção e Remoção

- A inserção em uma pilha sempre é chamada ***push***, enquanto a remoção é chamada ***pop***.

- Uma pilha *S* com até *M* elementos pode ser implementada com o uso de um **Array** estático de tamanho máximo *M*.

S

1	2	3	4	5	6	7	8
15	6	3	1	**8**			

S.topo = 5

Pilhas – Estruturas Básicas

- Para a pilha estática, teremos estruturas básicas semelhantes às das listas sequenciais.

```
1. registro NoPilhaSeq
2.      chave:inteiro
```

```
1. registro PilhaSeq
2.      maximo:inteiro,
3.      topo:inteiro,
4.      dados:Array<NoPilhaSeq>
```

Pilhas – Inserção

- No pior caso, o procedimento *push* tem complexidade da ordem de $\theta(1)$.

```
1. X -> NoPilhaSeq a ser inserido, pilha -> PilhaSeq
2. procedimento push(X, pilha)
3.       n = pilha.topo
4.       se n < pilha.maximo
5.           n = n + 1
6.           pilha.dados[n] = X
7.           pilha.topo = n
8.       senão
9.           imprimir("Overflow!")
```

Pilhas – Inserção

- Inserir 14, 2, 1, 5, nesta ordem:

S

1	2	3	4	5	6	7	8
15	6	3	1	**8**			

S.topo = 5

S

1	2	3	4	5	6	7	8
15	6	3	1	8			

S.topo = 6

Pilhas – Inserção

- Inserir 14, 2, 1, 5, nesta ordem: (Cont.)

S

1	2	3	4	5	6	7	8
15	6	3	1	8	**14**		

S.topo = 6

…

Ao tentar inserir o elemento 5, ocorrerá *overflow*.

S

1	2	3	4	5	6	7	8
15	6	3	1	8	14	2	**1**

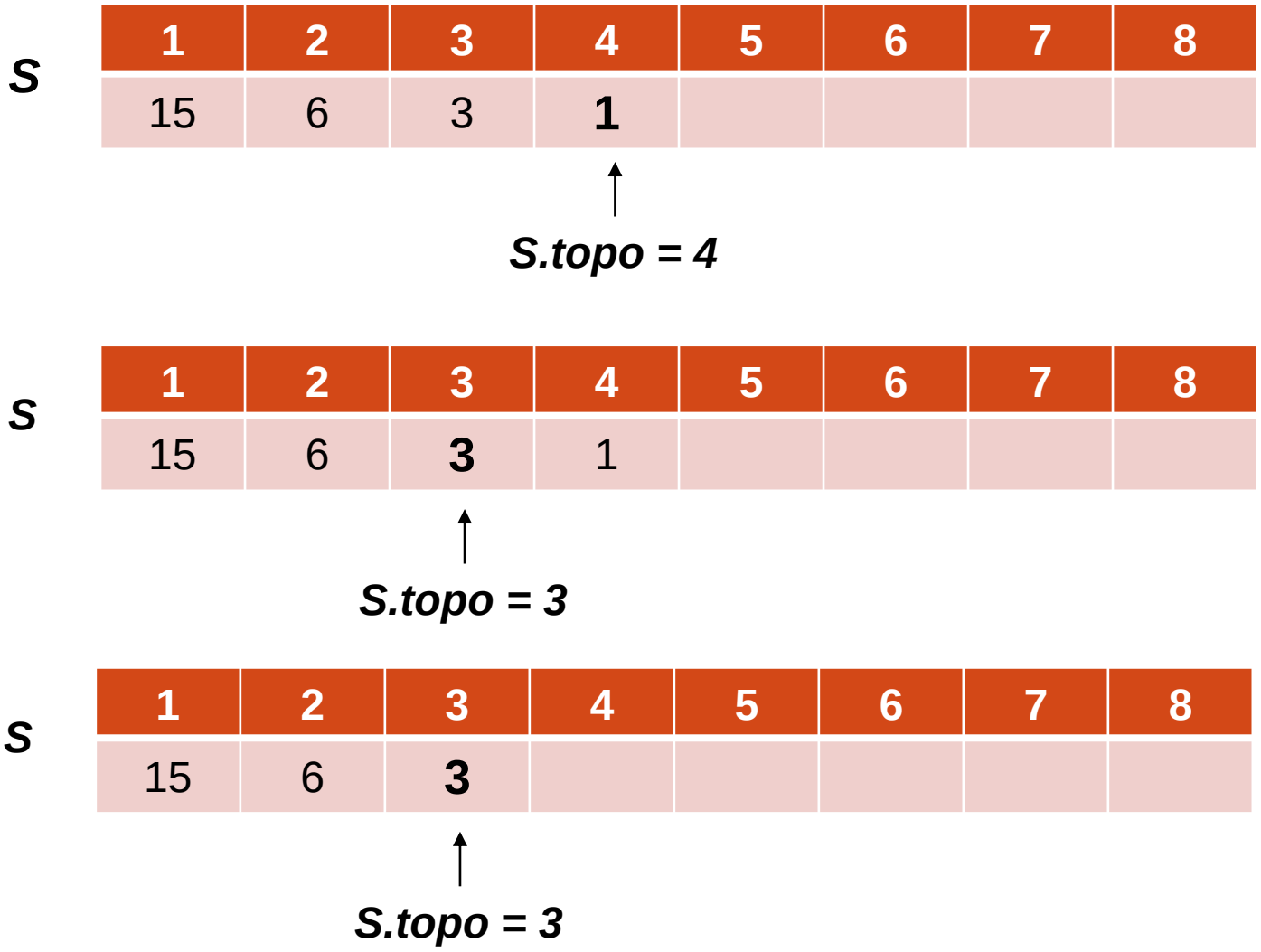
S.topo = 8

Pilhas – Remoção

- No pior caso, o procedimento *pop* tem complexidade da ordem de $\theta(1)$.

```
1.  //pilha -> PilhaSeq
2.  procedimento pop(pilha)
3.      removido = NIL
4.      n = pilha.topo
5.      se n != 0
6.          removido = pilha.dados[n]
7.          pilha.topo = n - 1
8.      senão
9.          imprimir("Underflow!")
10.     retorne removido
```

Pilhas – Remoção



Filas

- Para o caso de uma fila *Q* qualquer, a inserção ocorrerá em uma de suas extremidades (*Q.fim*), enquanto a remoção ocorrerá na outra extremidade (*Q.inicio*).

- Da mesma forma que ocorre em uma pilha, a operação de busca não será definida para as filas, uma vez que só suas extremidades podem ser acessadas.

- O primeiro elemento a ser inserido é sempre o primeiro elemento a ser removido (política **FIFO**, ou *first-in-first-out*).

Filas – Inserção e Remoção

- Nos exemplos a seguir, consideraremos uma fila com no máximo *M* – 1 elementos. O índice Q.fim sempre apontará para a **posição vazia** na qual o próximo objeto deverá ser inserido na fila.

- O desperdício de uma posição de memória será compensado pela facilidade de execução das operações de inserção e remoção sem que a estrutura de dados **degenere**.

- Quando uma fila *Q* está vazia temos *Q.inicio* = *Q.fim*.

- Quando *Q.inicio* = *Q.fim* + 1 ou *Q.inicio* = 1 e *Q.fim* = *M*, a fila está cheia.

- Tentativas de remoção em filas vazias resultarão em ***underflow***, enquanto tentativas de inserção em filas cheias resultarão em ***overflow***.

Filas – Estruturas de Básicas

```
1. registro NoFilaSeq
2.       chave:inteiro
```

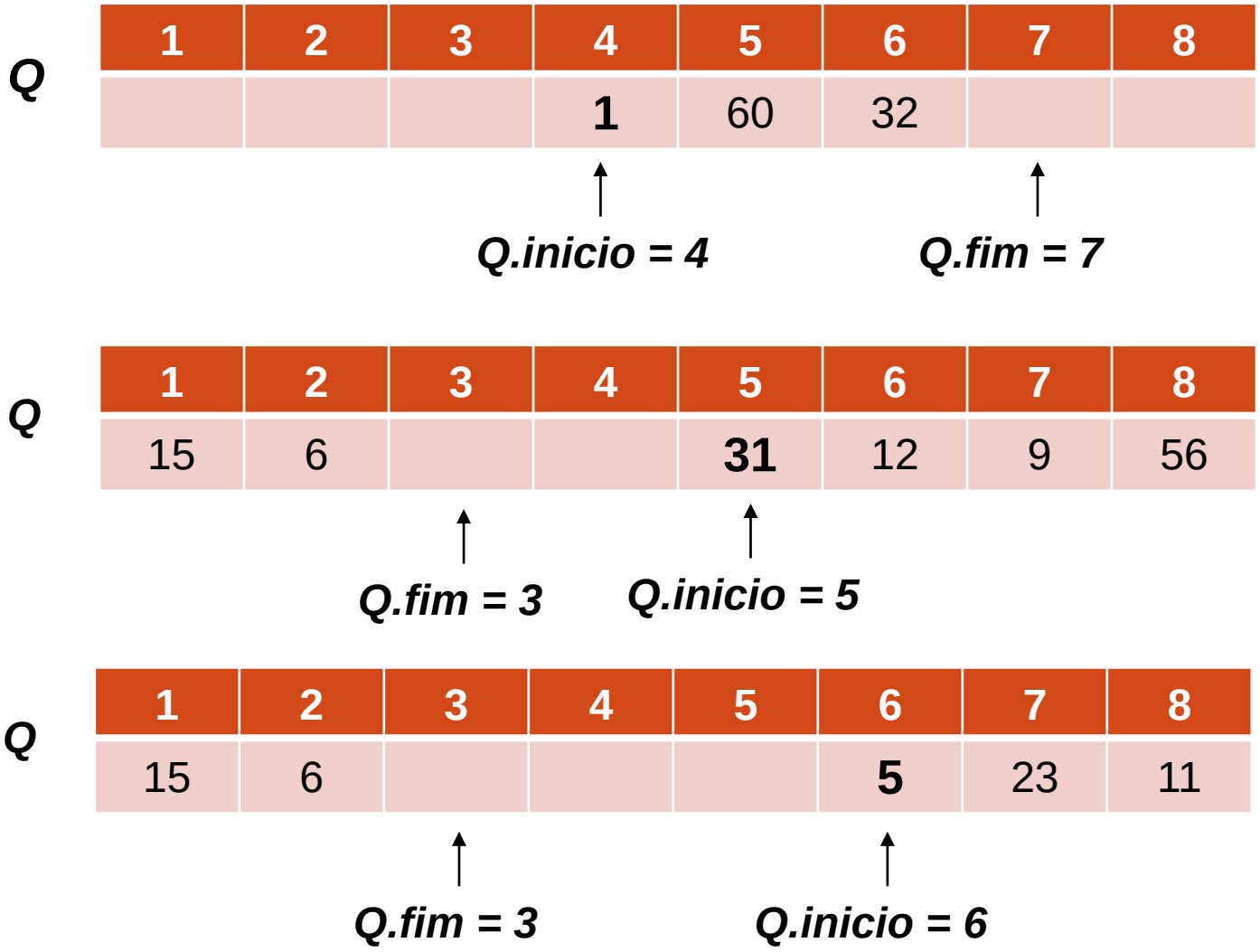
```
1. registro FilaSeq
2.      maximo:inteiro,
3.      inicio:inteiro,
4.      fim:inteiro,
5.      dados:Array<NoFilaSeq>
```

Filas – Inserção e Remoção

```
1.  //X -> NoFilaSeq a ser inserido, fila -> FilaSeq
2.  procedimento inserir(X, fila)
3.      se (fila.inicio == 1 e fila.fim == M) ou (fila.inicio == fila.fim + 1)
4.          imprimir("Overflow!")
5.      senão
6.          fila.dados[fila.fim] = X
7.          se fila.fim == M
8.              fila.fim = 1
9.          senão
10.             fila.fim = fila.fim + 1
```

```
1.  //fila -> FilaSeq
2.  procedimento remover(X, fila)
3.      removido = NIL
4.      se fila.inicio == fila.fim
5.          imprimir("Underflow!")
6.      senão
7.          removido = fila.dados[fila.inicio]
8.          se fila.inicio == M
9.              fila.inicio = 1
10.         senão
11.             fila.inicio = fila.inicio + 1
12.     retorne removido
```

Filas – Exemplo



Referências

- SZWARCFITER, J.; MARKENZON, L. Estruturas de Dados e seus Algoritmos, 3ª ed. Rio de Janeiro: LTC, 2010.

- CORMEN, H. T.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms, 3rd ed., *Boston: MIT Press*, 2009.

- FEOFILOFF, Paulo. Algoritmos em Linguagem C. Editora Campus/Elsevier, 2009.

Listas Lineares Sequenciais

Algoritmos e Estruturas de Dados

Prof. Dr. Luciano Demétrio Santos Pacífico

{luciano.pacifico@ufrpe.br}

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO