Algoritmos e Estrutura de Dados II

Pilha, Fila e Lista
(Alocação Dinâmica)

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso
Instituto de Engenharia

Roteiro

Objetivos

Esta aula tem como objetivos:

1. Apresentar os conceitos básicos sobre filas, pilhas e listas;
2. Explicitar as diferenças, vantagens e desvantagens de cada um;
3. Exemplificar os algoritmos por meio de pseudo-códigos.

Introdução

- Alocação Dinâmica é o processo de solicitar e utilizar memória durante a execução de um programa.
- Ela é utilizada para que um programa em C utilize apenas a memória necessária pra sua execução, sem desperdícios de memória.
- Um exemplo de desperdício de memória é quando um vetor de 1000 posições é declarado quando não se sabe, de fato, se as 1000 posiçoes serão necessárias.
- A alocação dinâmica de memória deve ser utilizada quando não se sabe quanto espaço de memória será necessário pra o armazenamento de algum ou alguns valores.

Vantagens e Desvantagens

- Vantagens
  - Ocupa espaço estritamente necessário
- Desvantagens
  - Custos usuais da alocação dinâmica (tempo de alocação, campos de ligação)

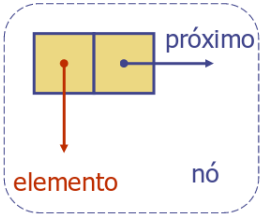
Estrutura Nodo

- A entidade elementar de uma estrutura de dados é o **nodo** (pronuncia-se nó).
- Um nodo é diferenciado pelo seu endereço relativo dentro da estrutura e pode ser constituído de um ou vários campos.
- Cada campo de um nodo armazena uma informação, um dado, que pode ser do tipo numérico, alfanumérico, lógicos, imagens, sons, enfim, qualquer informação que possa ser manipulada.
- Todos os nodos de uma mesma estrutura devem ter a mesma configuração.

Estrutura Nodo

Cada nodo armazena:

- Um elemento.
  - pode ser um inteiro, uma string, um vetor, um registro, não importa.
  - Aqui é denominado apenas **item**.
- Uma ligação para o próximo nodo.
  - Em termos de implementação, trata-se de um ponteiro do tipo nodo, o qual denominaremos **prox**.



Estrutura Nodo

Algoritmo 1: Nodo

```
1 início
2     registro {
3         Inteiro: item;
4         Ponteiro Nodo: prox;
5     } Nodo;
```

Implementação Dinâmica
Pilhas

Estrutura Pilha

A seguir, são apresentadas as caracteristicas da estrutura de dados Pilha:

- Existem dois ponteiros:
  - um aponta para o **topo** da Pilha.
  - o outro que aponta para o **fundo** da Pilha.
- No topo da Pilha existe um nodo vazio (denominado nodo **cabeça**).
- O ponteiro topo sempre aponta para o nodo cabeça.
- Quando a Pilha está vazia, fundo e topo apontam para o nodo cabeça.

Estrutura Pilha

Algoritmo 2: Pilha

```
1 início
2     registro {
3         Ponteiro Nodo: topo, fundo;
4     } Pilha;
```

Pilhas - Operações básicas

1. CriaPilhaVazia(S)
2. boolean PilhaVazia(S)
3. Empilha(S,x)
4. int Desempilha(S)
5. ApagaPilha(S);

Pilha - Implementação com ponteiros

Algoritmo 3: CriaPilhaVazia

Entrada: Pilha S.

```
1 início
2     S.topo ← ALOCA_NODO()
3     S.fundo ← S.topo
4     S.topo.prox ← NULL
```

Pilhas - Implementação com ponteiros

Algoritmo 4: PilhaVazia

Entrada: Pilha S.
Saída: Booleano (V ou F) indicando se S está vazia.

```
1 início
2     se (S.topo = S.fundo) então
3         retorna Verdadeiro
4     senão
5         retorna Falso
```

Pilhas - Implementação com ponteiros

Algoritmo 5: Empilha

Entrada: Pilha S, item x.

```
1 início
2     novo ← ALOCA_NODO()
3     novo.prox ← S.topo
4     S.topo.item ← x
5     S.topo ← novo
```

Pilhas - Implementação com ponteiros

Algoritmo 6: Desempilha

Entrada: Pilha S.
Saída: Item desempilhado.

```
1 início
2     se (PilhaVazia(S)) então
3         Imprima "Erro underflow: pilha vazia."
4     senão
5         aux ← S.topo
6         S.topo ← aux.prox
7         item ← aux.prox.item
8         DESALOCA_NODO(aux)
9         retorna item
```

Pilha - Implementação com ponteiros

Algoritmo 7: ApagaPilha

Entrada: Pilha S.

```
1 início
2     enquanto (NOT(PilhaVazia(S))) faça
3         // Ignora o elemento desempilhado.
4         Desempilha(S)
5     // Apaga o nodo cabeça.
6     DESALOCA_NODO(S.topo)
7     S.topo ← NULL
8     S.fundo ← NULL
```

Implementação Dinâmica
Filas

Estrutura Fila

A seguir, são apresentadas as caracteristicas da estrutura de dados Fila:

- Existem dois ponteiros:
  - um aponta para o **início** da Fila.
  - o outro que aponta para o **fim** da Fila.
- No início da Fila existe um nodo vazio (denominado nodo **cabeça**).
- O ponteiro início sempre aponta para o nodo cabeça.
- Quando a Fila está vazia, início e fim apontam para o nodo cabeça.

Estrutura Fila

Algoritmo 8: Fila

```
1 início
2     registro {
3         Ponteiro Nodo: início, fim;
4     } Fila;
```

Filas - Operações básicas

1. CriaFilaVazia(Q)
2. boolean FilaVazia(Q)
3. Enfileira(Q,x)
4. int Desenfileira(Q)
5. ApagaFila(Q)

Filas - Implementação com ponteiros

Algoritmo 9: CriaFilaVazia

Entrada: Fila Q.

```
1 início
2     Q.inicio ← ALOCA_NODO()
3     Q.fim ← Q.inicio
4     Q.inicio.prox ← NULL
```

Filas - Implementação com ponteiros

Algoritmo 10: FilaVazia

Entrada: Fila Q.
Saída: Booleano (V ou F) indicando se Q está vazia.

```
1 início
2     se (Q.inicio = Q.fim) então
3         retorna VERDADEIRO
4     senão
5         retorna FALSO
```

Filas - Implementação com ponteiros

Algoritmo 11: Enfileira

Entrada: Fila Q, item x.

```
1 início
2     Q.fim.prox ← ALOCA_NODO()
3     Q.fim ← Q.fim.prox
4     Q.fim.item ← x
5     Q.fim.prox ← NULL
```

Filas - Implementação com ponteiros

Algoritmo 12: Desenfileira

Entrada: Fila Q.
Saída: Item desenfileirado.

```
1  início
2      se ( FilaVazia(Q) ) então
3          Imprima "Erro underflow: fila esta vazia."
4      senão
5          aux ← Q.inicio
6          Q.inicio ← Q.inicio.prox
7          item ← Q.inicio.item
8          aux.prox ← NULL
9          DESALOCA_NODO(aux)
10         retorna item
```

Filas - Implementação com ponteiros

Algoritmo 13: ApagaFila

Entrada: Fila Q.

```
1 início
2     enquanto (NOT(FilaVazia(Q))) faça
3         // Ignora o elemento desenfileirado.
4         Desenfila(Q)
5     // Apaga o nodo cabeça.
6     DESALOCA_NODO(Q.inicio)
7     Q.inicio ← NULL
8     Q.fim ← NULL
```

Implementação Dinâmica
Listas

Listas

A estrutura e funcionamento de uma lista encadeada é semelhante a dos vetores, porém, com algumas diferenças importantes:

- Os elementos dos vetores podem ser acessados diretamente. Nas listas, os nós são acessados sequencialmente, pelos ponteiros;
- O espaço previsto para vetores já é alocado e fica reservado. Nas listas, o espaço de memória é alocada conforme necessário.

Estrutura Lista

A seguir, são apresentadas as caracteristicas da estrutura de dados Lista:

- Existem dois ponteiros:
  - um aponta para o **primeiro** nodo da Lista.
  - o outro que aponta para o **último** nodo da Lista.
- Diferentemente da Pilha e da Fila, não existe o nodo **cabeça**.
- Quando a Lista está vazia, os ponteiros primeiro e último apontam para NULL.

Estrutura Lista

Algoritmo 14: Lista

```
1 início
2     registro {
3         Ponteiro Nodo: primeiro, último;
4     } Lista;
```

Estrutura Nodo

Algoritmo 15: Nodo

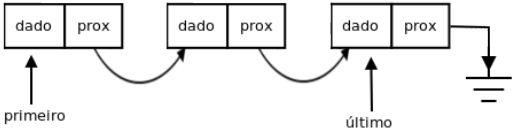
```
1 início
2     registro {
3         Inteiro: item;
4         Ponteiro Nodo: prox;
5     } Nodo;
```

Listas

Tipos de Listas

- Simplesmente encadeada;
- Duplamente encadeada;
- Circular.

Lista Simplesmente Encadeada



Lista Simplesmente Encadeada

Operações básicas

1. CriaListaVazia(L)
2. boolean ListaVazia(L)
3. int ListaBuscar(L, x)
4. int ListaBuscarPosição(L, p)
5. ListaInserirInício(L, x)
6. ListaInserirFinal(L, x)
7. ListaInserirPosição(L, x, p)
8. ApagarLista(L)
9. int ListaRemoverInício(L)
10. int ListaRemoverFinal(L)
11. int ListaRemoverPosição(L, p)

Algoritmo 16: CriaListaVazia

Entrada: Lista L.

```
1 início
2     L.primeiro ← NULL
3     L.último ← NULL
```

Algoritmo 17: ListaVazia

Entrada: Lista L.
Saída: Booleano (V ou F) indicando se L está vazia.

```
1 início
2     se (L.primeiro = NULL) então
3         retorna Verdadeiro
4     senão
5         retorna Falso
```

Algoritmo 18: ListaBuscar

Entrada: Lista L, item x.
Saída: Nodo que contém x ou NULL caso não encontrado.

```
1 início
2     aux ← L.primeiro
3     enquanto (aux ≠ NULL) AND (aux.item ≠ x) faça
4         aux ← aux.prox
5     retorna aux
```

Algoritmo 19: ListaBuscarPosição

Entrada: Lista L, posição p.
Saída: Nodo que se encontra na posição p ou NULL caso não encontrado.

```
1 início
2     aux ← L.primeiro
3     c ← 1
4     enquanto (aux ≠ NULL) AND (c < p) faça
5         aux ← aux.prox
6         c ← c + 1
7     retorna aux
```

Lista Simplesmente Encadeada

Operações básicas

Algoritmo 20: ListaInserirInício

Entrada: Lista L, item x.

```
1 início
2     novo ← ALOCA_NODO()
3     novo.item ← x
4     novo.prox ← L.primeiro
5     L.primeiro ← novo
6     // Verifica se a lista está vazia.
7     se (L.último = NULL) então
8         L.último ← L.primeiro
```

Lista Simplesmente Encadeada

Operações básicas

Algoritmo 21: ListaInserirFinal

Entrada: Lista L, item x.

```
1  início
2      novo ← ALOCA_NODO()
3      novo.item ← x
4      novo.prox ← NULL
5      // Verifica se a lista está vazia.
6      se (L.primeiro = NULL) então
7          L.primeiro ← novo
8          L.último ← L.primeiro
9      senão
10         L.último.prox ← novo
11         L.último ← novo
```

Lista Simplesmente Encadeada

Operações básicas

- O algoritmo a seguir insere um nodo com item x na posição p da lista.
- Não verifica se existe um nodo na posição p, simplesmente considera que o nodo existe.
- A seguir, uma descrição dos passos do algoritmo:
  1. Primeiramente, verifica se p é a primeira posição.
     - Caso sim, invoca a função ListaInserirInicio.
     - Caso contrário, busca o nodo na posição $p - 1$, ou seja, o nodo anterior a p.
  2. Em seguida, insere o novo nodo entre os nodos nas posições $p - 1$ (denominado *anterior*) e p (denominado *proximo*).
  3. Por fim, verifica se o nodo inserido na posição p é o último nodo da lista. Caso sim, atualiza o ponteiro último.

Lista Simplesmente Encadeada

Operações básicas

Algoritmo 22: ListaInserirPosição

Entrada: Lista L, item x, posição p.

```
1  início
2      // Verifica se o nodo deve ser inserido no início.
3      se (p = 1) então
4          ListaInserirInício(L, x)
5      senão
6          novo ← ALOCA_NODO()
7          novo.item ← x
8          // Busca o nodo na posição anterior a p.
9          anterior ← ListaBuscarPosição(L,p − 1)
10         // Insere o novo nodo entre os nodos
11         // anterior e posterior.
12         posterior ← anterior.prox
13         anterior.prox ← novo
14         novo.prox ← posterior
15         // Verifica se o nodo na posição p − 1 é o último,
16         // ou seja, seu posterior é NULL.
17         se (posterior = NULL) então
18             L.último ← novo
```

Algoritmo 23: ApagarLista

Entrada: Lista L.

```
1 início
2     aux ← L.primeiro
3     enquanto (L.primeiro ≠ NULL) faça
4         L.primeiro ← L.primeiro.prox
5         DESALOCA_NODO(aux)
6         aux = L.primeiro
```

Lista Simplesmente Encadeada

Operações básicas

Algoritmo 24: ListaRemoverInício

Entrada: Lista L, item x.

```
1  início
2      // Verifica se a lista está vazia.
3      se (ListaVazia(L)) então
4          Imprima "Erro underflow: lista vazia"
5      senão
6          aux ← L.primeiro
7          x ← aux.item
8          L.primeiro ← L.primeiro.prox
9          // Verifica se a lista possui apenas um nodo.
10         se (L.primeiro = NULL) então
11             // Primeiro e Último apontam para NULL.
12             L.último ← NULL
13         aux.prox ← NULL
14         DESALOCA_NODO(aux)
15         retorna x
```

Lista Simplesmente Encadeada

Operações básicas

Algoritmo 25: ListaRemoverFinal

Entrada: Lista L
Saída: Item removido

```
1  início
2      // Verifica se a lista está vazia.
3      se (ListaVazia(L)) então
4          Imprima "Erro underflow: lista vazia"
5      senão
6          // Verifica se a lista possui um único elemento.
7          se (L.primeiro = L.último) AND (NOT(ListaVazia(L))) então
8              aux ← L.primeiro
9              L.primeiro ← NULL
10             L.último ← NULL
11         // Caso contrário, a lista possui
12         // pelo menos dois elementos.
13         senão
14             // Busca o penú ltimo elemento.
15             anterior ← L.primeiro
16             enquanto (anterior.prox ≠ L.último) faça
17                 anterior ← anterior.prox
18             aux ← L.último
19             anterior.prox ← NULL
20             L.último ← anterior
21         x ← aux.item
22         aux.prox ← NULL
23         DESALOCA_NODO(aux)
24         retorna x
```

Lista Simplesmente Encadeada

Operações básicas

Algoritmo 26: ListaRemoverPosição

Entrada: Lista L, posição p
Saída: Item removido

```
1  início
2      // Verifica se a lista está vazia.
3      se (ListaVazia(L)) então
4          Imprima "Erro underflow: lista vazia"
5      senão
6          // Verifica se o nodo a ser removido é o primeiro.
7          se (p = 1) então
8              retorna ListaRemoverInicio(L)
9          senão
10             // Busca o nodo na posição anterior a p.
11             anterior ← ListaBuscarPosição(L,p − 1)
12             // Remove o nodo na posição p, apontado por aux.
13             aux ← anterior.prox
14             posterior ← aux.prox
15             anterior.prox ← posterior
16             // Verifica se o nodo a ser removido é o último.
17             se (aux = L.último) então
18                 L.último ← anterior
19             x ← aux.item
20             aux.prox ← NULL
21             DESALOCA_NODO(aux)
22             retorna x
```

Implementação Dinâmica
Listas Duplamente Encadeadas

Estrutura Lista

A seguir, são apresentadas as características da Lista Duplamente Encadeada que são idênticas à Lista Simplesmente Encadeada:

- Existem dois ponteiros:
  - um aponta para o **primeiro** nodo da Lista.
  - o outro que aponta para o **último** nodo da Lista.
- Não existe o nodo **cabeça**.
- Quando a Lista está vazia, os ponteiros primeiro e último apontam para NULL.

Lista Duplamente Encadeada

Estrutura Nodo

Diferentemente do nodo presente na Lista Simplesmente Encadeada, na lista Duplamente Encadeada cada nodo armazena:

- Um elemento.
  - pode ser um inteiro, uma string, um vetor, um registro (não importa), também denominado **item**.
- Uma ligação para o próximo nodo.
- Uma ligação para o nodo anterior.

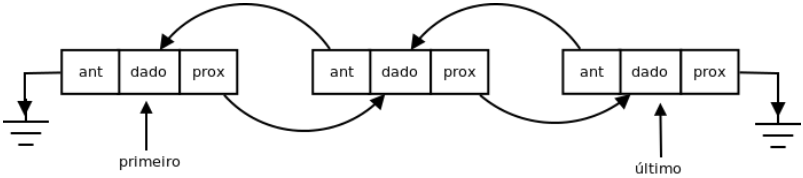
Lista Duplamente Encadeada

Estrutura Nodo

Algoritmo 27: Nodo

```
1 início
2     registro {
3         Inteiro: item;
4         Ponteiro Nodo: prox, ant;
5     } Nodo;
```

Lista Duplamente Encadeada



Lista Duplamente Encadeada

Operações básicas

1. ListaInserirInício(L, x)
2. ListaInserirFinal(L, x)
3. ListaInserirPosição(L, x, p)
4. int ListaRemoverInício(L)
5. int ListaRemoverFinal(L)
6. int ListaRemoverPosição(L, p)

Lista Duplamente Encadeada - Implementação

Algoritmo 28: ListaInserirInício

Entrada: Lista L, item x.

```
1  início
2      novo ← ALOCA_NODO()
3      novo.item ← x
4      novo.prox ← L.primeiro
5      // Verifica se a lista está vazia.
6      se (ListaVazia(L)) então
7          L.último ← novo
8      senão
9          L.primeiro.ant ← novo
10     L.primeiro ← novo
11     L.primeiro.ant ← NULL
```

Lista Duplamente Encadeada

Operações básicas

Algoritmo 29: ListaInserirFinal

Entrada: Lista L, item x.

```
1  início
2      novo ← ALOCA_NODO()
3      novo.item ← x
4      novo.prox ← NULL
5      // Verifica se a lista está vazia.
6      se (ListaVazia(L)) então
7          L.primeiro ← novo
8      senão
9          L.último.prox ← novo
10     novo.ant ← L.último
11     L.último ← novo
```

Lista Duplamente Encadeada

Operações básicas

Algoritmo 30: ListaInserirPosição

Entrada: Lista L, item x, posição p.

```
1  início
2      // Verifica se o nodo deve ser inserido no início.
3      se (p = 1) então
4          ListaInserirInício(L, x)
5      senão
6          novo ← ALOCA_NODO()
7          novo.item ← x
8          // Busca o nodo na posição anterior a p.
9          anterior ← ListaBuscarPosição(L,p − 1)
10         // Insere o novo nodo entre os nodos
11         // anterior e posterior.
12         posterior ← anterior.prox
13         anterior.prox ← novo
14         novo.ant ← anterior
15         novo.prox ← posterior
16         // Verifica se o nodo na posição p − 1 é o último,
17         // ou seja, seu posterior é NULL.
18         se (posterior = NULL) então
19             L.último ← novo
20         senão
21             posterior.ant ← novo
```

Lista Duplamente Encadeada

Operações básicas

Algoritmo 31: ListaRemoverInício

Entrada: Lista L, item x.

```
1  início
2      // Verifica se a lista está vazia.
3      se (ListaVazia(L)) então
4          Imprima "Erro underflow: lista vazia"
5      senão
6          aux ← L.primeiro
7          x ← aux.item
8          L.primeiro ← L.primeiro.prox
9          // Verifica se a lista possui apenas um nodo.
10         se (L.primeiro = NULL) então
11             // Primeiro e Último apontam para NULL.
12             L.último ← NULL
13         senão
14             L.primeiro.ant ← NULL
15         aux.prox ← NULL
16         aux.ant ← NULL
17         DESALOCA_NODO(aux)
18         retorna x
```

Lista Duplamente Encadeada

Operações básicas

Algoritmo 32: ListaRemoverFinal

Entrada: Lista L
Saída: Item removido

```
1  início
2      // Verifica se a lista está vazia.
3      se (ListaVazia(L)) então
4          Imprima "Erro underflow: lista vazia"
5      senão
6          // Verifica se a lista possui um único elemento.
7          se (L.primeiro = L.último) AND (NOT(ListaVazia(L))) então
8              aux ← L.primeiro
9              L.primeiro ← NULL
10             L.último ← NULL
11         // Caso contrário, a lista possui
12         // pelo menos dois elementos.
13         senão
14             anterior ← L.último.ant
15             aux ← L.último
16             anterior.prox ← NULL
17             L.último ← anterior
18         x ← aux.item
19         aux.prox ← NULL
20         aux.ant ← NULL
21         DESALOCA_NODO(aux)
22         retorna x
```

Lista Duplamente Encadeada

Operações básicas

Algoritmo 33: ListaRemoverPosição

Entrada: Lista L, posição p
Saída: Item removido

```
1  início
2      // Verifica se a lista está vazia.
3      se (ListaVazia(L)) então
4          Imprima "Erro underflow: lista vazia"
5      senão
6          // Verifica se o nodo a ser removido é o primeiro.
7          se (p = 1) então
8              retorna ListaRemoverInicio(L)
9          senão
10             // Busca o nodo na posição anterior a p.
11             anterior ← ListaBuscarPosição(L,p − 1)
12             // Remove o nodo na posição p, apontado por aux.
13             aux ← anterior.prox
14             posterior ← aux.prox
15             anterior.prox ← posterior
16             // Verifica se o nodo a ser removido é o último.
17             se (aux = L.último) então
18                 L.último ← anterior
19             senão
20                 posterior.ant ← anterior
21             x ← aux.item
22             aux.prox ← NULL
23             aux.ant ← NULL
24             DESALOCA_NODO(aux)
25             retorna x
```

Lista Duplamente Encadeada

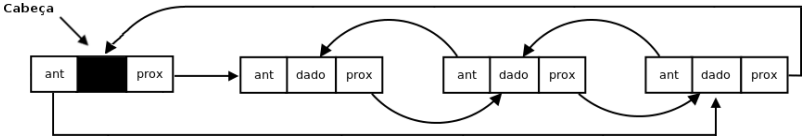
É possivel simplificar?

Lista Duplamente Encadeada

É possivel simplificar? Sim!
Solução: adicionar uma sentinela: o nodo cabeça.

Implementação Dinâmica
Listas Circularers

Lista Circular



Lista Circular - Implementação

Algoritmo 34: ListaCircularBuscar

Entrada: Lista C, item x.
Saída: Nó da lista que contém o item x ou NULL caso não encontrado.

```
1 início
2     aux ← C.cabeça.prox
3     enquanto (aux ≠ C.cabeça) AND (aux.item ≠ x) faça
4         aux ← aux.prox
5     retorna aux
```

Lista Circular - Implementação

Algoritmo 35: ListaCircularInserir

Entrada: Lista circular C, item x.

```
1 início
2     novo ← ALOCA_NODO()
3     novo.item ← x
4     novo.prox ← C.cabeça.prox
5     C.cabeça.prox.ant ← novo
6     C.cabeça.prox ← novo
7     novo.ant ← C.cabeça
```

Lista Circular - Implementação

Algoritmo 36: ListaCircularRemover

Entrada: Lista C, item x.
Saída: V ou F.

```
1  início
2      aux ← ListaCircularBuscar(C, x)
3      se (aux ≠ C.cabeça) então
4          aux.ant.prox ← aux.prox
5          aux.prox.ant ← aux.ant
6          aux.prox ← NULL
7          aux.ant ← NULL
8          DESALOCA_NODO(aux)
9          retorna Verdadeiro
10     senão
11         retorna Falso
```

Conclusão

Material de apoio

Animações das operações disponíveis em http://www.ime.usp.br/~nelio/ensino/2002-1/ed/

Livro Base

Projeto de Algoritmos - Nívio Ziviani - Capítulo 3, Seção 3.2 e 3.3

Algoritmos e Estrutura de Dados II

Pilha, Fila e Lista
(Alocação Dinâmica)

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso
Instituto de Engenharia