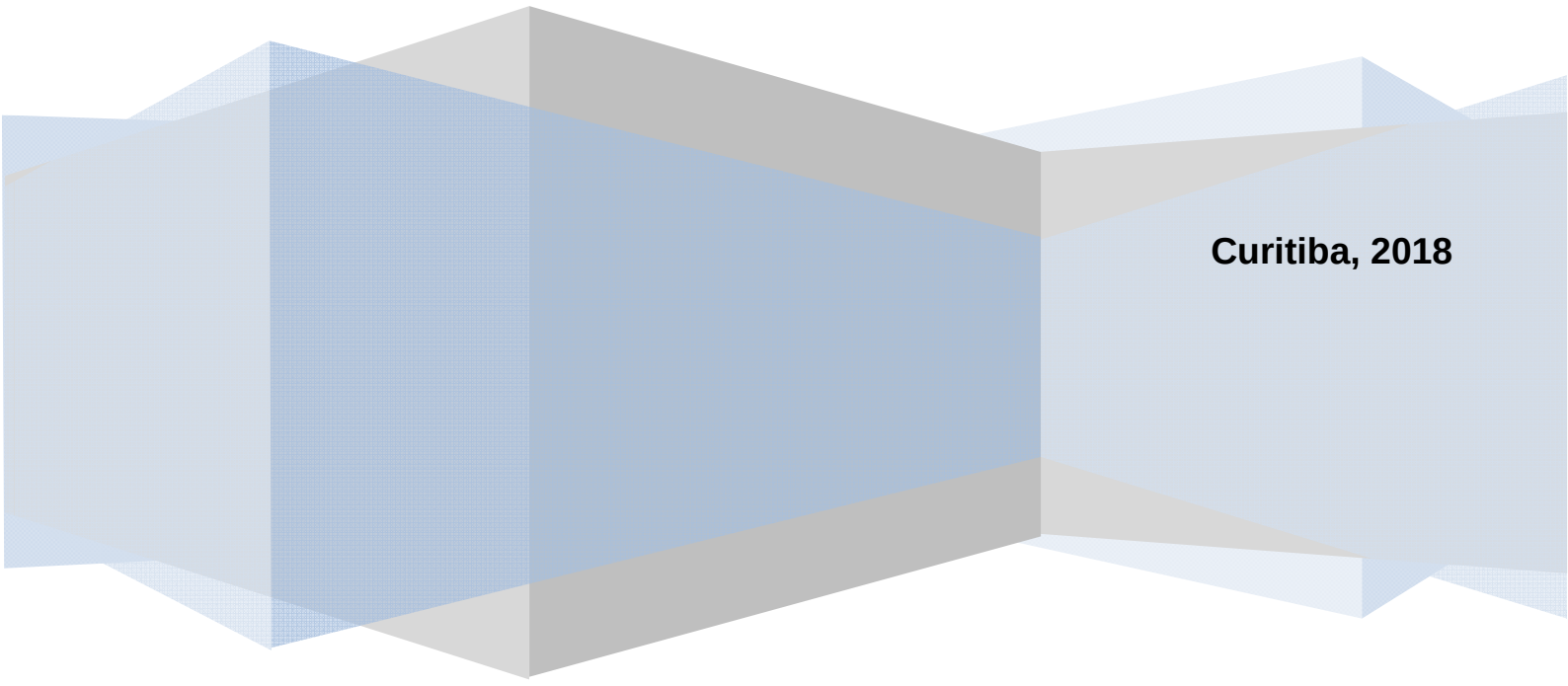
Introdução às estruturas de dados

Cicero Aparecido Bezerra



Curitiba, 2018

SUMÁRIO

1 SOBRE O AUTOR

Possui graduação em Informática pela Universidade do Vale do Rio dos Sinos (1992), mestrado em Engenharia de Produção pela Universidade Federal de Santa Catarina (2001), doutorado em Engenharia de Produção pela Universidade Federal de Santa Catarina (2007) e estágio pós-doutoral em Gestão Estratégica da Informação e do Conhecimento pela Pontifícia Universidade Católica do Paraná (2012). Atualmente é professor Associado nível II da Universidade Federal do Paraná. Tem experiência em profissional em desenvolvimento, implantação e gestão de Sistemas de Informação e, Processos de Produção. Enquanto docente, leciona disciplinas alinhadas ao desenvolvimento de Sistemas de Informação e Análise de Dados. Como pesquisador, tem voltado sua atenção aos Métodos e Técnicas de Análise de Dados, aplicados à Gestão do Conhecimento e Inovação.

2 APRESENTAÇÃO

O material ao qual vocês estão tendo acesso apresenta os principais conceitos de estrutura de dados. Buscou-se desenvolver o material com uma linguagem acessível e com ilustrações que pudessem esclarecer os tópicos.

Na primeira parte, apresentamos as principais estruturas de dados (listas, pilhas e filas), bem como as operações mais comumente efetuadas. Em seguida tratamos de alguns métodos de ordenação de dados para, em seguida, abordar algoritmos de busca de dados.

Espero que este material possa ser útil na compreensão básica dos conceitos e aplicações das estruturas de dados.

Bons estudos...

3 ESTRUTURAS DE DADOS

Conceitos

Por definição, temos um conjunto básico de dados primitivos (inteiro, real, caracter e lógico). Quando agrupamos estes dados, formamos uma **estrutura**. Este agrupamento pode resultar em vetores (matrizes unidimensionais), matrizes (com mais de uma dimensão) ou registros.

Objetivos

O estudo das estruturas de dados é necessário para que possamos identificar e desenvolver modelos matemáticos que resolvam problemas abstratos, bem como a resolução de problemas práticos que envolvam estruturas de dados.

3.1 Listas lineares

Conceitos

A maneira mais básica de se agrupar dados é a lista. Matematicamente, uma lista é um conjunto **L** (e_1, e_2, e_3, . . . , e_n) com as seguintes propriedades para n > 0:

- e_1 é o primeiro elemento de **L**;
- e_n é o último elemento de **L**;
- um elemento e_k é precedido pelo elemento e_{k-1} e seguido por e_{k+1}.

Caso n = 0, dizemos que a lista é vazia. A ideia de lista é a unidimensionalidade dos elementos, ou seja, temos sempre um elemento a frente de outro. Com este conceito podemos afirmar com certeza absoluta que a lista tem um início e fim bem determinados.

Tipos de listas

Considerando as operações de inserção, remoção e pesquisa de elementos restritas às extremidades da lista, temos seguintes tipos especiais de listas:

- **Pilha:** inserções, remoções e pesquisa de elementos são realizadas a partir de uma única extremidade da lista, ou seja, o último elemento que entrou na lista, será o primeiro elemento a sair.
- **Fila:** as inserções são realizadas em um extremo e as remoções e pesquisa são realizadas no outro extremo, ou seja, o primeiro elemento que entrou na lista, será o primeiro elemento a sair.

Cada tipo de lista pode ser implementada de forma **sequencial** (quando cada célula da estrutura possui apenas o elemento contido nela), **simplesmente encadeada** (quando cada célula está dividida em duas partições, onde uma partição irá guardar o seu elemento e a outra partição guardará o endereço da célula seguinte) ou **duplamente encadeada** (quando cada célula está dividida em três partições, onde a primeira partição irá guardar o endereço da célula anterior, a segunda partição guardará seu elemento e a outra partição guardará o endereço da célula seguinte).

Uma lista pode ocupar a memória do computador de forma **estática** (quando, de antemão, já sabemos seu tamanho) ou **dinâmica** (quando o programa é capaz de criar novas varáveis durante sua execução).

As listas apresentam diversas particularidades, que serão estudadas no seu devido tempo, bem como outros tipos de estruturas de dados.

3.2 Pilhas sequenciais

Conceito

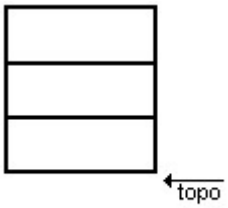
Uma pilha é um tipo especial de lista linear em que as operações de inserção e remoção são realizadas na mesma extremidade, a qual chamaremos **topo**.

Desta forma, cada vez que um novo elemento deverá ser inserido na pilha, devemos executar esta inserção colocando-o sobre o último elemento da pilha, consequentemente alterando, seu topo. Para a remoção, somente o elemento localizado no topo da pilha, pode ser retirado, onde altera-se também o topo.

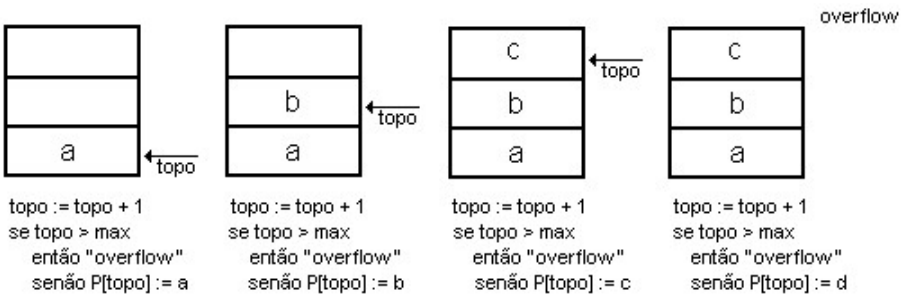
Operações

Em uma pilha, temos basicamente as seguintes operações:

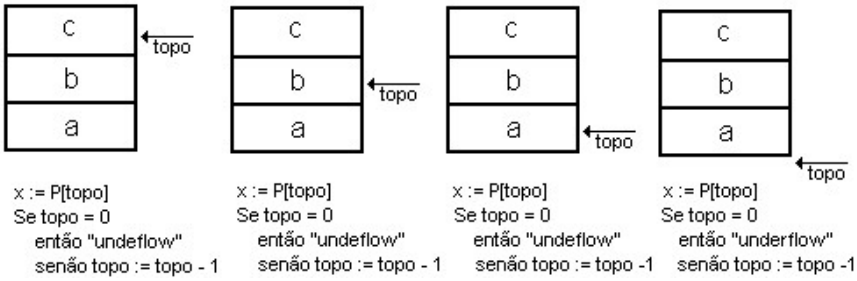
- **Inicialização:** dizemos que uma pilha P está vazia, quando seu topo for igual a zero.



- **Inserção:** podemos inserir elementos até que o topo atinja o tamanho máximo da pilha.



- **Remoção:** podemos retirar elementos da pilha, até que o topo atinja o valor zero.

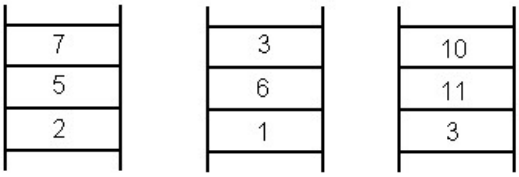


A operação de "remoção", na verdade apenas desloca o topo, não retirando fisicamente o elemento contido na posição atual do topo, a não ser que inicializemos o elemento a cada "retirada".

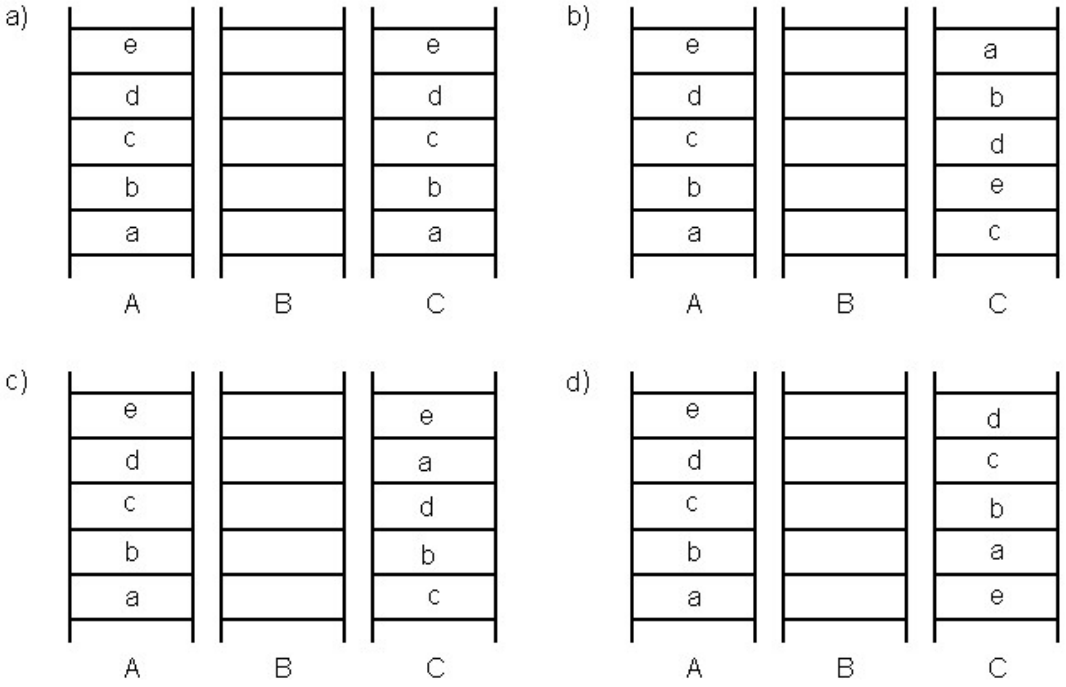
Exercícios

1. Faça um algoritmo para converter um número decimal inteiro para a base binária e depois mostrar o resultado.

2. Faça um algoritmo para somar os valores contidos nos topos de 2 pilhas e armazená-los na posição correspondente em uma terceira pilha.



3. Mostre as seqüências de operações de inserção e retirada para atender as seguintes situações:



4. Faça um algoritmo para ler uma frase e imprimi-la com as palavras invertidas. Exemplo: a frase "sistemas de informacao" ficará impressa "sametsis ed oacamrofni".

5. Faça um algoritmo para verificar se uma cadeia de caracteres é a mesma quando lida de direita para a esquerda ou vice-versa. Exemplo: "aoxomoxoa", "subinoonibus".

6. Faça um algoritmo para inserir um elemento em uma pilha em qualquer posição. Deve-se levar em consideração que a inserção poderá ser impossível, uma vez que 1 elemento a mais poderá causar "overflow".

7. Existem situações onde é possível armazenar 2 pilhas em um único vetor, de modo que as inserções de uma pilha começam na primeira posição do vetor e as inserções da outra pilha começam na última posição e ambas avançam em direção ao centro do vetor. Esta situação otimiza o espaço ocupado por pilhas de alocação sequencial (ou estática). Faça um algoritmo para a inserção de um elemento em uma das pilhas. Deve-se levar em consideração que ambas as pilhas podem estar cheias.

Pilha A					Pilha B				
a	B	c	d	...	α	β	μ	π	θ
1	2	3	4	...	n-4	n-3	n-2	n-1	n

8. Faça um algoritmo para classificar uma pilha de números inteiros em ordem decrescente de tamanho.

3.3 Pilhas encadeadas

Conceitos

Até então trabalhamos com o conceito de **pilha sequencial**, ou seja, os elementos estavam dispostos de forma **contínua** na memória. Existem situações onde os elementos não estão armazenados desta forma: existem "espaços" entre eles ou a disposição lógica é diferente da disposição física. Para que possamos manipular pilhas nesta situação, é necessário portanto, sabermos qual o endereço do próximo elemento da pilha.

8	marrom
7	púrpura
6	azul
5	preto
4	cinza
3	branco
2	amarelo
1	vermelho

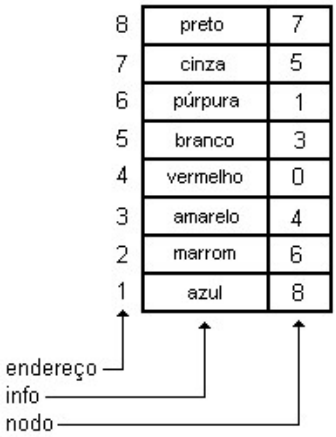
Pilha Sequencial

8	preto	7
7	cinza	5
6	púrpura	1
5	branco	3
4	vermelho	0
3	amarelo	4
2	marrom	6
1	azul	8

Pilha Encadeada

Elementos

Uma pilha encadeada possui elementos característicos, sem os quais torna-se impossível sua manipulação.



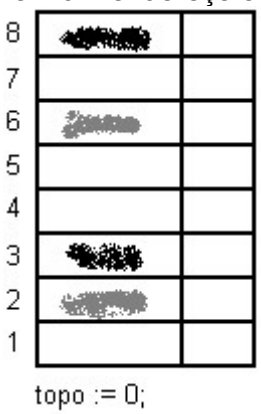
- **Endereço:** é a posição **física** de onde o elemento se encontra.
- **Info:** é a partição do elemento da pilha onde a informação está armazenada.
- **Nodo:** é a partição do elemento da pilha onde se encontra o **endereço** do próximo elemento **lógico** da pilha.

Por se tratar de uma pilha, temos ainda o **topo** que indica o **endereço** do último elemento da pilha (o primeiro a sair).

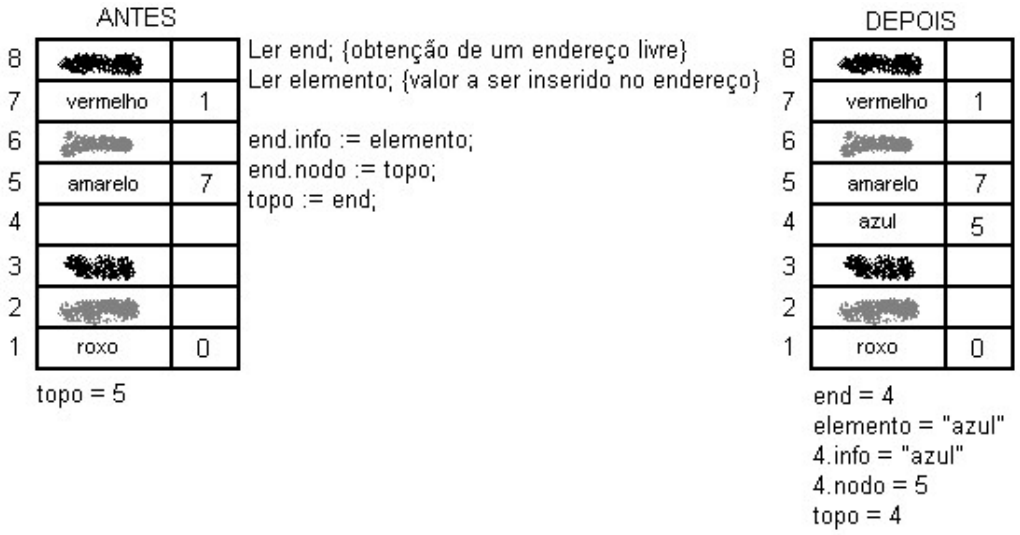
9. Operações

Assim como uma pilha de alocação sequencial, temos as operações de inicialização da pilha, inserção de um elemento e remoção de um elemento:

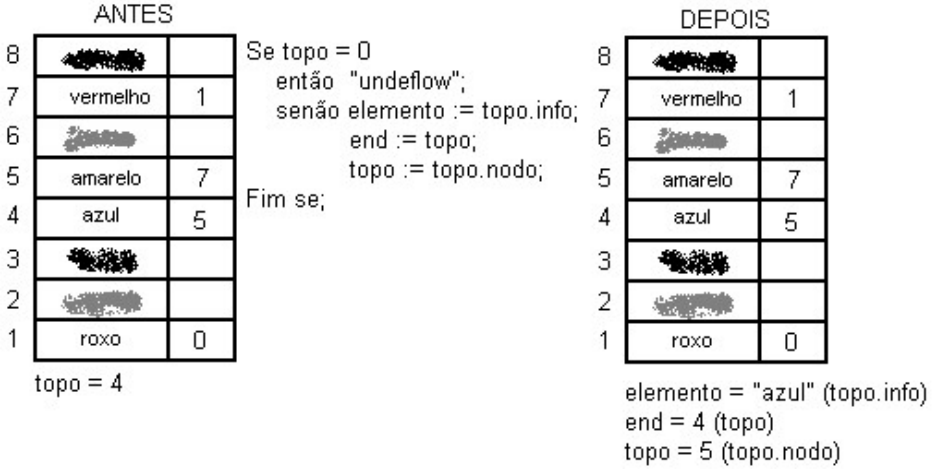
- **Inicialização:** dizemos que uma pilha P está vazia, quando seu topo for igual a zero, ou seja, não existe nenhum endereço ocupado pela pilha.



- **Inserção:** podemos inserir elementos até que o topo atinja o tamanho máximo da pilha.



- **Remoção:** podemos retirar elementos da pilha, até que o topo atinja o valor zero.



A operação de "remoção", na verdade apenas desloca o topo, não retirando fisicamente o elemento contido na posição atual do topo, a não ser que inicializemos o elemento a cada "retirada".

Exercícios

1. Faça um algoritmo para preencher todos os endereços disponíveis de uma pilha encadeada, a partir das informações de uma pilha sequencial.

2. Faça um algoritmo para retirar todas as informações constantes em uma pilha encadeada.

3. Sabe-se que um vetor pode armazenar mais de 1 pilha encadeada. Imagine que em um vetor existam 2 pilhas encadeadas com o mesmo número de elementos. Faça um algoritmo para somar as informações constantes em cada endereço das pilhas e armazene-os em uma terceira pilha encadeada, no mesmo vetor, verificando a possibilidade de "overflow".

4. Faça um algoritmo para inserir um elemento em uma posição qualquer de uma pilha encadeada.

5. Faça um algoritmo para inverter a posição dos elementos de uma pilha encadeada.

3.4 Filas

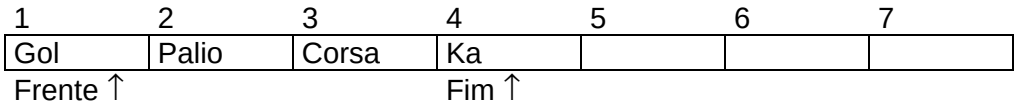
Conceitos

Uma fila é uma estrutura de dados cujas inserções são realizadas em um extremo da lista (que chamaremos **fim**) e cujas retiradas são realizadas no outro extremo da lista (que chamaremos **frente**). Uma operação de inserção é aquela onde novos elementos serão adicionados à fila e uma operação de retirada é aquela onde poderemos acessar, apagar, consultar o elemento.

A fila é representada tal e qual a conhecemos no dia-a-dia. A primeira pessoa que entra na fila é a primeira a ser atendida. Enquanto esta pessoa não for atendida, a próxima pessoa se posicionará atrás da primeira e assim sucessivamente. Quando a pessoa que esta na primeira posição é atendida, a posição de primeiro é ocupada pela pessoa que esta na segunda posição e assim sucessivamente.

6. Elementos

Uma fila possui elementos característicos, sem os quais torna-se impossível sua manipulação.



Tamanho = 7
Frente = 1
Fim = 4

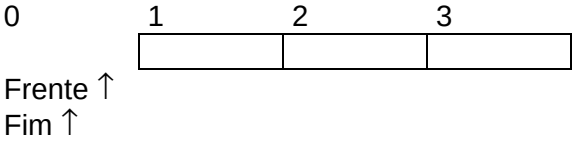
- **Frente:** é a posição que indica o próximo elemento a ser acessado.
- **Fim:** é a posição que indica o próximo elemento a ser inserido.
- **Tamanho:** é o número máximo de posições permitidas na fila.

Operações

Com nas estruturas de dados estudadas até então, temos as operações de inicialização de uma fila, inserção de um elemento e retirada de um elemento.

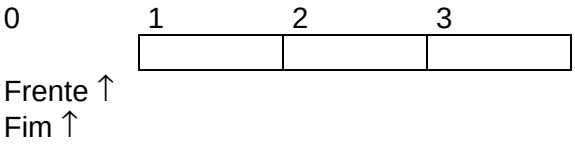
- **Inicialização:** dizemos que uma fila F está vazia, quando sua **frente** e seu **fim** for igual a zero, ou seja, não existe nenhuma posição da estrutura ocupada por algum elemento.

Tamanho = 3
Frente = 0
Fim = 0

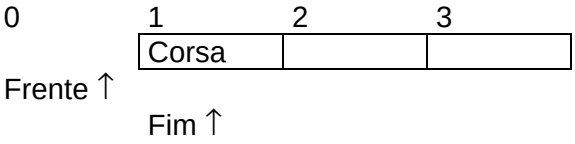


- **Inserção:** podemos inserir elementos até que o **fim** seja igual ao **tamanho**. Inserção de elementos além do tamanho da fila, configura-se um caso de *overflow.*

Tamanho = 3
Frente = 0
Fim = 0

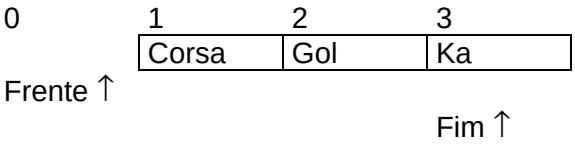


Se **fim** = **tamanho**
Então "overflow";
Senão **fim** := **fim** + 1;
Fila.**fim**:= "Corsa";
Fim se;



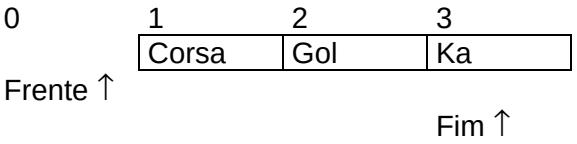
Se **fim** = **tamanho**
Então "overflow";
Senão **fim** := **fim** + 1;
Fila.**fim**:= "Gol";
Fim se;

Se **fim** = **tamanho**
Então "overflow";
Senão **fim** := **fim** + 1;
Fila.**fim**:= "Ka";
Fim se;

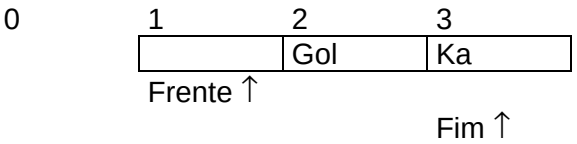


- **Remoção:** podemos retirar elementos da fila, até que **frente** seja igual ao **fim**. Retirada de elementos além desta situação, configura-se um caso de *undeflow*.

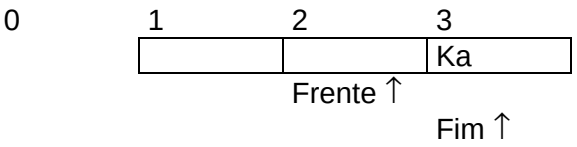
Tamanho = 3
Frente = 0
Fim = 3



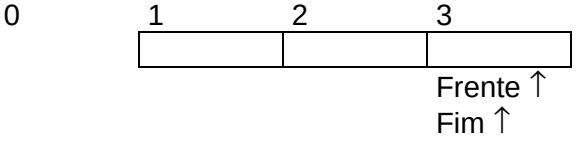

```
Se frente = fim
      Então      "underflow";
      Senão      frente := frente + 1;
                   Fila.frente:= "     ";
                   Se frente = fim
                         Então  fim := 0;
                                      frente := 0;
                         Fim se;
Fim se;
```

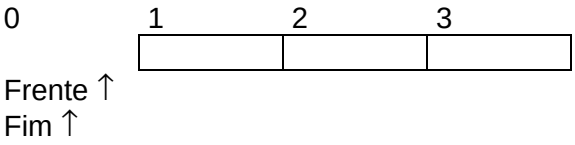



```
Se frente = fim
      Então      "underflow";
      Senão      frente := frente + 1;
                   Fila.frente:= "     ";
                   Se frente = fim
                         Então  fim := 0;
                                      frente := 0;
                         Fim se;
Fim se;
```




```
Se frente = fim
      Então      "underflow";
      Senão      frente := frente + 1;
                   Fila.frente:= "     ";
                   Se frente = fim
                         Então  fim := 0;
                                      frente := 0;
                         Fim se;
Fim se;
```





Exercícios

1. Faça um algoritmo para preencher uma fila de **tamanho** = 10. O algoritmo deve inicializar a fila e controlar situações de *overflow*.

2. Faça um algoritmo para retirar todos as informações constantes em uma fila de **tamanho** = 10. Partir do princípio de que a fila está completamente preenchida e controlar situações de *underflow*.

3. Faça um algoritmo para manipular uma fila de **tamanho** = 10, de modo que ao digitar um número, se este for par, deve ser inserido na fila e se for ímpar deve-se retirar um elemento da fila. O algoritmo deve inicializar a fila e controlar situações de *overflow* e *underflow*.

4. Faça um algoritmo para somar os elementos de 2 filas A e B, de **tamanho** = 8.354.782 e armazenar o resultado em uma terceira fila C de mesmo tamanho. Partir do princípio de que existem elementos nas filas A e B. O algoritmo deve inicializar a fila C.

5. Faça um algoritmo para manipular duas filas A e B de **tamanho** = 10.548, de modo que ao digitar um número, se este for par, deve ser inserido na fila A e se for ímpar deve-se retirar um elemento da fila A e inseri-lo na fila B. O algoritmo deve inicializar as filas e controlar situações de *overflow* e *underflow*.

6. Faça um algoritmo para inserir elementos pares em uma fila A e elementos ímpares 100 em uma fila B. Ambas as filas tem tamanho máximo de 1.000 posições. Quando o elemento for igual a 0, deve-se interromper as inserções e criar uma fila C, onde a primeira posição será ocupada pelo maior elemento das primeiras posições das filas A e B; o segundo elemento de C receberá o menor; a terceira posição de C receberá o maior elemento das segundas posições das filas A e B; a quarta posição receberá o menor elemento entre das segundas posições de A e B e assim sucessivamente. O algoritmo deve considerar situações de *underflow* e *overflow*.

Fila A	8	10	6	2	4
Fila B		9	5	7	3
Fila C	6	2	7	3	4

3.5 Fila circular

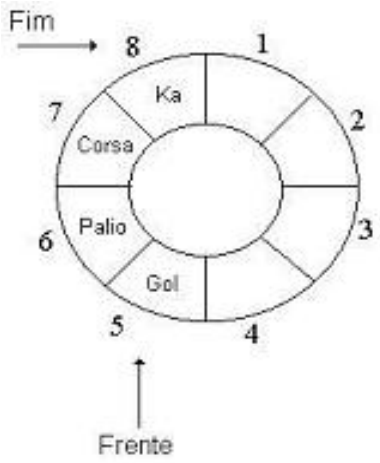
Conceitos

Conforme as operações de inserção e retirada em uma fila vão ocorrendo aleatoriamente, pode-se chegar à seguinte situação:

Tamanho = 8
Frente = 5
Fim = 8

Ou seja, a partir deste momento, só é possível a operação de retirada de elementos. Apesar de existirem endereços disponíveis na fila (1, 2, 3 e 4), o ponteiro que indica o **fim** da fila já chegou ao **tamanho** máximo da mesma.

Ora, existem espaços livres e não podemos inserir? Certamente ocorreu uma disfunção no conceito de fila. Para solucionar esta anomalia, representa-se a fila de forma circular:



Desta forma, a próxima inserção será efetuada na primeira posição livre e irá parar quando uma próxima posição estiver ocupada.

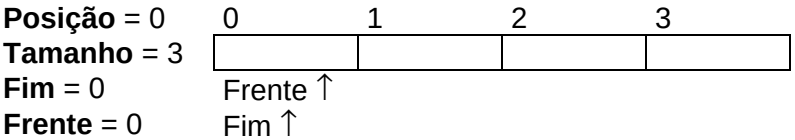
Continuamos, porém, trabalhando com os mesmos elementos de uma fila de representação sequencial, acrescentando mais um elemento:

- **Frente:** é a posição que indica o próximo elemento a ser acessado.
- **Fim:** é a posição que indica o próximo elemento a ser inserido.
- **Tamanho:** é o número máximo de posições permitidas na fila.
- **Posição:** é o número de posições ocupadas na fila.

7. Operações

Com nas estruturas de dados estudadas até então, temos as operações de inicialização de uma fila, inserção de um elemento e retirada de um elemento.

- **Inicialização:** dizemos que uma fila F está vazia, quando sua **frente** e seu **fim** for igual a 0 (zero).



- **Inserção:** podemos inserir elementos até que a **posição** seja igual ao **tamanho** da fila. Inserção de elementos além desta situação, configura-se um caso de *overflow.*

Se **posição** < **tamanho**

```
        Então   Se fim = tamanho
                        Então fim := 0;
                Fim se;
                fim := fim + 1;
                info.fim := "nome de carro"
                posição := posição + 1;
        Senão   mostrar "fila cheia";
Fim se;
```

Posição = 1
Tamanho = 3
Fim = 1
Frente = 0

0	1	2	3
	Corsa		

Frente ↑ Fim ↑

Posição = 2
Tamanho = 3
Fim = 2
Frente = 0

0	1	2	3
	Corsa	Palio	

Frente ↑ Fim ↑

Posição = 3
Tamanho = 3
Fim = 3
Frente = 0

0	1	2	3
	Corsa	Palio	Ka

Frente ↑ Fim ↑

A partir deste momento não conseguimos inserir mais elementos na fila, pois existem tantas posições ocupadas quanto o tamanho da fila permite.

- **Remoção:** podemos retirar elementos da fila, até que a **posição** seja igual a 0 (zero). Retirada de elementos além desta situação, configura-se um caso de *undeflow*.

```
Se posição > 0
        Então   Se frente = tamanho
                        Então frente := 0;
                Fim se;
                frente := frente + 1;
                info.frente := " "
                posição := posição - 1;
        Senão   mostrar "fila vazia";
                        frente := 0;
                        fim := 0;
Fim se;
```

Posição = 2
Tamanho = 3
Fim = 3
Frente = 1

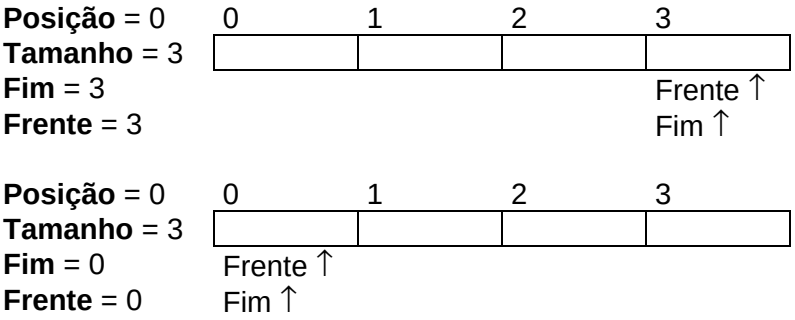
0	1	2	3
		Palio	Ka

Frente ↑ Fim ↑

Posição = 1
Tamanho = 3
Fim = 3
Frente = 2

0	1	2	3
			Ka

Frente ↑ Fim ↑



Exercícios

1. Utilizando o teste de mesa efetuado nos exemplos de inserção e remoção de elementos de uma fila circular, verifique a possibilidade de executar as seguintes operações **exatamente na ordem apresentada**:

   a) inserção, inserção, inserção, remoção e inserção;
   b) inserção, inserção, inserção, remoção, inserção, remoção e remoção;
   c) inserção, inserção, remoção, inserção, remoção, inserção, inserção e inserção.

2. Faça um algoritmo para manipular uma fila circular, de forma a preencher posições, quando um elemento digitado for maior ou igual a 100 e disponibilizar posições quando um elemento digitado for menor que 100. Partir do princípio de que a fila possui tamanho igual 10 e que está vazia. Controlar situações de *underflow* e *overflow*.

3. Faça um algoritmo para manipular duas filas circulares A e B de **tamanho** = 10, de modo que ao digitar um número, se este for par, deve ser inserido na fila A e se for ímpar deve-se retirar um elemento da fila A e inseri-lo na fila B. O algoritmo deve inicializar as filas e controlar situações de *overflow* e *underflow*.

4. Faça um algoritmo para inserir elementos positivos em uma fila circular A e elementos negativos em uma fila circular B. Ambas as filas tem tamanho máximo de 1.000 posições. Quando o elemento for igual a 0, ou ocorrer situação de *overflow* em uma das filas, deve-se interromper as inserções e criar uma fila circular C, que será preenchida com a soma dos elementos contidos nas posições das filas A e B.

Fila A	8	10	9	3	4
Fila B	- 9	-5	-7	-3	
Fila C	-1	5	2	0	4

4 ORDENAÇÃO DE DADOS

A ordenação de dados (representados pelas informações armazenadas em vetores) corresponde ao uso econômico da memória disponível, ou seja, para ordenarmos um vetor, a melhor maneira é aquela na qual não ocorra a necessidade de criarmos outro vetor, e sim executarmos as permutas dos elementos, no vetor inicial.

É importante frisar que, dentre as inúmeras técnicas de ordenação de vetores existentes, utilizaremos aquelas conhecidas por métodos diretos, em função da baixa complexidade. Existem métodos mais rápidos, porém mais complexos que serão vistos oportunamente.

4.1 Ordenação por inserção direta

Este método é muito utilizado pelos jogadores de carta. Consiste, basicamente, em dividir os elementos em uma seqüência destino e uma seqüência origem. A cada passo, iniciando-se a pesquisa a partir do segundo elemento e incrementando-se de 1 em 1, o último elemento da seqüência vai sendo retirado e transferido para a seqüência destino, inserido na posição apropriada.

Exemplo:

```
program direta;
uses crt;
var vetor: array [0..5] of integer;
    ind, aux: byte;
    memo: integer;

procedure tela;
begin
    clrscr;
    gotoxy(4,5); write ('+---+---+---+---+---+');
    gotoxy(4,6); write ('|   |   |   |   |   |');
    gotoxy(4,7); write ('+---+---+---+---+---+');
end;

procedure preenche;
var col: byte;
begin
    col:=6;
    for ind:=0 to 5 do
        begin
            vetor[ind]:=' ';
            gotoxy(col,6); readln (vetor[ind]);
            col:=col + 4;
        end;
end;

procedure mostra;
var col: byte;
begin
    col:=6;
    for ind:=0 to 5 do
        begin
```

```
            gotoxy(col,6); write (vetor[ind]);
            col:=col + 4;
        end;
end;

{programa principal}
begin
    ind:=0;
    aux:=0;
    memo:=0;

    tela;
    preenche;

    {ordenacao direta}
    for aux:= 1 to 5 do
        begin
            memo:=vetor[aux];
            ind:=aux-1;
            while ((ind >= 0) and (vetor[ind] > memo) do
                begin
                    vetor[ind + 1]:=vetor[ind];
                    ind:=ind-1;
                end;
            vetor[ind + 1]:=memo;
        end;

    mostra;
    readkey;
end.
```

O resultado da inserção direta é ilustrado abaixo:

início	6	3	4	5	9	8
aux = 1	3	6	4	5	9	8
aux = 2	3	4	6	5	9	8
aux = 3	3	4	5	6	9	8
aux = 4	3	4	5	6	9	8
aux = 5	3	4	5	6	8	9

4.2 Ordenação por inserção binária

Este método é derivado da inserção direta, porém torna-se mais rápido visto que determina o ponto correto de inserção. Consiste, basicamente, em amostrar a seqüência destino no seu ponto central, continuando a bissecção até encontrar o ponto correto de inserção.

Exemplo:

```
{ordenacao insercao binaria}
for ind:=2 to 6 do
    begin
        memo:=vetor[ind];
        esq:=1;
        dir:=ind;
        while esq < dir do
            begin
                meio:=(esq+dir) div 2;
                if vetor[meio]<=memo
                    then esq:=meio+1
                    else dir:=meio;
            end;
        for aux:=ind downto dir+1 do
            vetor[aux]:=vetor[aux-1];
        vetor[dir]:=memo;
    end;
```

Exercício:

Ilustre o método da inserção binária.

4.3 Ordenação por seleção direta

Este método consiste no seguinte princípio:

1º. selecionar o menor elemento da seqüência;
2º. trocá-lo com o primeiro elemento da seqüência;
3º. repetir estas operações, envolvendo os elementos restantes sucessivamente, até restar somente 1 elemento (que será o maior deles).

Exemplo:

```
{ordenacao selecao direta}
for ind:=1 to 6 do
    begin
        menor:=ind;
        memo:=vetor[ind];
        for aux:=ind+1 to 6 do
            if vetor[aux] < memo
                then begin
                        menor:=aux;
                        memo:=vetor[menor];
                    end;
        vetor[menor]:=vetor[ind];
        vetor[ind]:=memo;
    end;
```

O resultado da seleção direta é ilustrado abaixo:

início	44	55	12	42	94	18	06	67
I = 2	06	55	12	42	94	18	44	67
i = 3	06	12	55	42	94	18	44	67
i = 4	06	12	18	42	94	55	44	67
i = 5	06	12	18	42	94	55	44	67
i = 6	06	12	18	42	44	55	94	67
i = 7	06	12	18	42	44	55	94	67
i = 8	06	12	18	42	44	55	67	94

4.4 Bubblesort (ordenação por bolhas)

Este método consiste em efetuar varreduras repetidas sobre o vetor, deslocando-se a cada passo, para sua extremidade esquerda, o menor valor dos elementos do conjunto restante.

Exemplo:

```
{bubblesort}
for ind:=2 to 6 do
    for aux := 6 downto ind do
        if vetor[aux-1] > vetor[aux]
            then begin
                    memo := vetor[aux-1];
                    vetor[aux-1] := vetor[aux];
                    vetor[aux] := memo;
                end;
```

O nome “ordenação por bolhas”, vem da ideia de que, se o vetor for visualizado na posição vertical e, se com o uso da imaginação, os elementos forem bolhas em um tanque de água, com densidades proporcionais ao valor das respectivas chaves, então cada varredura resultaria na ascensão de uma bolha para o seu nível apropriado.

índice	1	2	3	4	5	6	7
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

5 PESQUISA DE DADOS

Da mesma forma que a ordenação de dados, a pesquisa e a recuperação de dados corresponde ao uso econômico da memória disponível para a localização do elemento.

5.1 Busca em tabela

Este método representa a forma mais simples para pesquisar a ocorrência de um elemento. Consiste em varrer uma tabela, do início ao fim, comparando cada elemento da tabela com o elemento procurado.

Exemplo:

```
Ler dadoProcurado;
Achou = "falso";
Para índice = 1 até n faça
    Se (tabela[índice] = dadoProcurado)
        Então Achou = "verdadeiro";
              Posição = índice;
    Fim se;
Fim para;
Se Achou  = "verdadeiro"
    Então mostrar "Dado na posição", posição;
    Senão mostrar "Dado não encontrado";
Fim se;
```

Tal algoritmo apresenta um certo desperdício de processamento, visto que, mesmo que encontrarmos o elemento em qualquer posição diferente da última, a comparação irá seguir até o final.

5.2 Busca em tabela melhorado

Apresenta uma melhoria em relação ao algoritmo anterior, visto que sua execução é interrompida tão logo o elemento seja encontrado.

Exemplo:

```
Ler dadoProcurado;
Achou = "falso";
Procura = "verdadeiro";
Índice = 0;
Enquanto (procura = "verdadeiro") faça
    Índice = índice + 1;
    Se (índice > tamanho)
        Então Procura = "falso";
        Senão Procura = (tabela[índice] ≠ dadoProcurado);
    Fim se;
Fim enquanto;
Se (índice ≤ tamanho)
    Então achou = "verdadeiro";
Fim se;
```

```
Se Achou  = "verdadeiro"
    Então mostrar "Dado na posição", índice;
    Senão mostrar "Dado não encontrado";
Fim se;
```

O algoritmo apresentado mostra uma certa ineficiência, visto que, para cada valor de <índice>, o algoritmo deve fazer duas comparações: a primeira para saber se <procura> é verdadeiro e a segunda para testar o valor de <índice>.

5.3 Busca em tabela com sentinela

Este algoritmo realiza a busca com apenas uma comparação, porém para que sua execução seja adequada é necessário acrescer a tabela de mais uma posição, inserindo o elemento procurado nesta posição. Desta forma é necessário apenas um teste.

1	2	3	4	5	6
50	11	32	98	23	dadoProcurado

Exemplo:

```
Ler dadoProcurado;
Achou = "falso";
Índice = 1;
Tabela[final] = dadoProcurado;
Enquanto (tabela[índice] ≠ tabela[final]) faça
    Índice = índice + 1;
Fim enquanto;
Achou = (índice ≠ final);
Se Achou = "verdadeiro"
    Então mostrar "Dado na posição", índice;
    Senão mostrar "Dado não encontrado";
Fim se;
```

5.4 Pesquisa binária

Nos algoritmos anteriores, percebe-se que, mesmo que o elemento não se encontre na tabela ocorrerá uma pesquisa (sequencial) a todos os elementos. Em relação a este problema, o algoritmo da busca binária, é mais eficiente, porém é necessário que todos os elementos estejam ordenados.

Seu funcionamento consiste, primeiramente, em considerar que o elemento procurado esteja no meio da tabela. Se este elemento for maior que o elemento procurado, podemos garantir que o elemento procurado está na segunda metade da tabela. O processo é repetido até que o elemento seja encontrado.

Exemplo:

```
Ler dadoProcurado;
Achou = "falso";
Início = 1;
Fim = tamanho;
Meio = (início + fim) div 2;
Enquanto (dadoProcurado ≠ tabela[meio] e (início ≠ fim)) faça
    Se (dadoProcurado > tabela[meio])
        Então início = meio + 1;
        Senão fim = meio;
    Fim se;
    Meio = (início + fim) div 2;
Fim enquanto;
Achou = (dadoProcurado = tabela[meio]);
Se Achou = "verdadeiro"
    Então mostrar "Dado na posição", índice;
    Senão mostrar "Dado não encontrado";
Fim se;
```

6 BIBLIOGRAFIA

CANTU, Marcos. **Dominando o Delphi 6**. São Paulo: Makron Books, 2001.

MORAES, Celso Roberto. **Estrutura de Dados e Algoritmos**. São Paulo: Berkeley, 2001.

PEREIRA, Silvio do Lago. **Estruturas de Dados Fundamentais: conceitos e aplicações**. São Paulo : Érica, 1996.

PREISS, Bruno R. **Estrutura de Dados**. Rio de Janeiro: Campus, 2000.

SZWARCFITER, Jayme Luiz. **Estruturas de Dados e seus Algoritmos**. Rio de Janeiro: LTC, 1994.

TEIXEIRA, Silvio; PACHECO, Xavier. **Delphi 5 – guia do desenvolvedor**. Rio de Janeiro: Campus, 2000.

VELOSO, Paulo et al. **Estruturas de Dados**. Rio de Janeiro : Campus, 1986.

WIRTH, Niklaus. **Algoritmos e Estruturas de Dados.** Rio de Janeiro: PHB, 1989.

ZIVIANI, Nivio. **Projeto de Algoritmos**. São Paulo : Pioneira, 1999.