

Estruturas de Dados

Fila

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

paulo_regis@uvanet.br

Fila

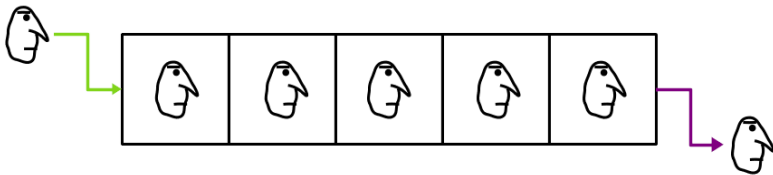
Fundamentos

Definição do TAD Fila

Implementação

Aplicação

- Fila é uma lista em que as inserções são feitas num extremo, denominado final, e as remoções são feitas no extremo oposto, denominado início.
- Devido a essa política de acesso, os itens de uma fila são removidos na mesma ordem em que foram inseridos, ou seja, o primeiro a entrar é o primeiro a sair. Por isso, as filas também são denominadas listas **FIFO** (*First-In/First-Out*).

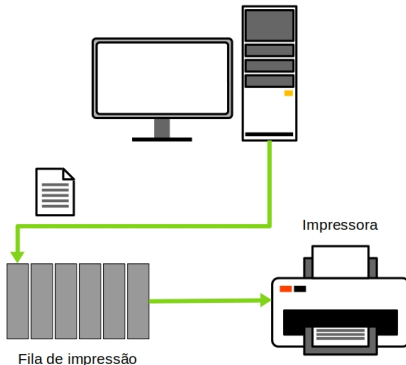


- A principal propriedade de uma fila é a sua capacidade de **manter a ordem** de uma sequência. Essa propriedade é útil em várias aplicações em computação.

Exemplo 1

Em um sistema operacional, cada solicitação de impressão de documento feita pelo usuário é inserida no final de uma fila de impressão.

Então, quando a impressora fica livre, o gerenciador de impressão atende à próxima solicitação de impressão, removendo-a do início dessa fila.

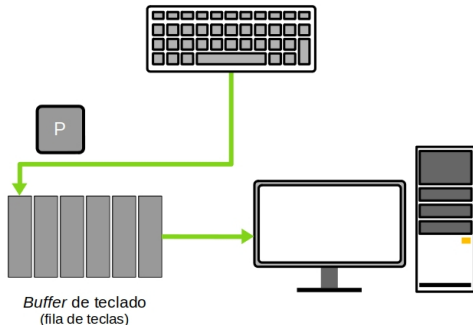


- A principal propriedade de uma fila é a sua capacidade de **manter a ordem** de uma sequência. Essa propriedade é útil em várias aplicações em computação.

Exemplo 2

Uma fila também é usada num sistema operacional para gerenciar a entrada de dados via teclado.

À medida que as teclas são pressionadas pelo usuário, os caracteres correspondentes são inseridos numa área de memória chamada buffer de teclado. Então, quando um caractere é lido por um programa, o primeiro caractere inserido no buffer de teclado é removido e devolvido como resposta.



- Uma fila F suporta as seguintes operações:
 - criação de uma fila vazia,
 - destruição de uma pilha,
 - verificação de fila vazia,
 - verificação de fila cheia,
 - inserção de um elemento no final da fila,
 - acesso e remoção do elemento no início da fila,
 - acesso ao elemento no início da fila sem sua remoção,
 - acesso ao elemento no final da fila sem sua remoção.

Código 1: Queue.h

```
1  typedef struct Queue Queue;  
2  
3  Queue *Queue_alloc(int size);  
4  void Queue_free(Queue *q);  
5  int Queue_isEmpty(Queue *q);  
6  int Queue_isFull(Queue *q);  
7  void Queue_push(Queue *q, int item);  
8  int Queue_pop(Queue *q);  
9  int Queue_begin(Queue *q);  
10 int Queue_end(Queue *q);
```

- Para exemplificar o uso de filas em programação, vamos criar um programa que verifica se uma cadeia é *palíndroma*.

Uma cadeia é palíndroma se ela é igual à sua inversa (ignorando-se os espaços).

- Para comparar as cadeias, basta percorrer a cadeia original, da esquerda para a direita, sem os espaços, inserindo simultaneamente em uma fila e em uma pilha cada letra encontrada.
- Comparando-se as letras removidas da fila e da pilha, podemos determinar se a cadeia original é ou não palíndroma.

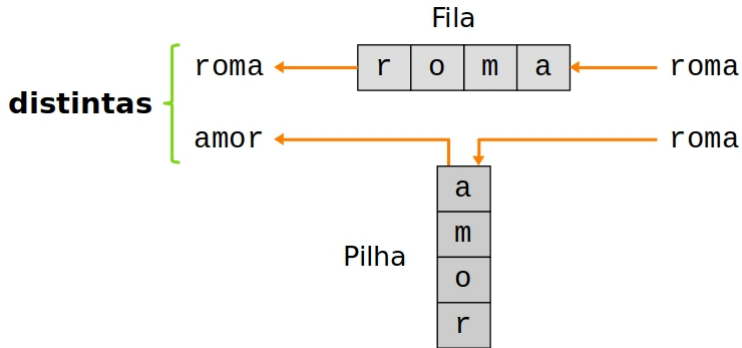


Figura 1: Comparação entre uma cadeia e sua inversa.

Cadeia palíndroma

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "Queue.h"
5  #include "Stack.h"
6  #define SIZE 32
7
8  int main() {
9      int i;
10     char strA[SIZE], strB[SIZE], strC[SIZE];
11     Queue *q = Queue_alloc(SIZE);
12     Stack *s = Stack_alloc(SIZE);
13
14     printf("Cadeia: ");
15     scanf("%[^\n]", strA);
16
17     for (i = 0; strA[i]; i++)
18         if (strA[i] != ' ') {
19             Queue_push(q, strA[i]);
20             Stack_push(s, strA[i]);
21     }
```

```
22
23     memset(strB, '\0', SIZE);
24     memset(strC, '\0', SIZE);
25
26     for (i = 0; !Stack_isEmpty(s); i++) {
27         strB[i] = Queue_pop(q);
28         strC[i] = Stack_pop(s);
29     }
30
31     if (strcmp(strB, strC) == 0)
32         printf("Is a palindrome chain (%s == %s)\n", strB, strC);
33     else
34         printf("Isn't a palindrome chain (%s != %s)\n", strB, strC);
35
36     Queue_free(q);
37     Stack_free(s);
38
39     return 0;
40 }
```

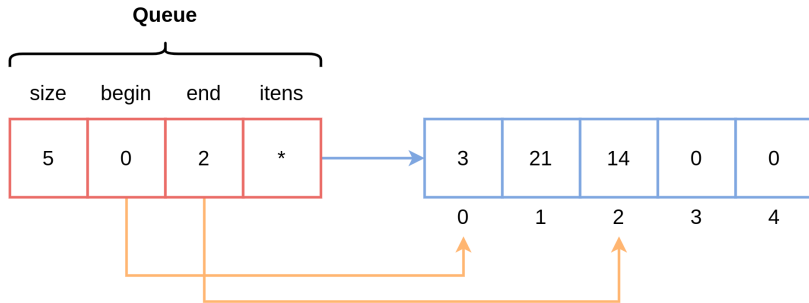


Figura 2: Representação de uma estrutura de fila com capacidade 5 e armazenando 3 itens.

- Para melhor aproveitamento de espaço no vetor $F \rightarrow \text{elements}$, vamos simular que esse vetor é circular, como na Figura abaixo.

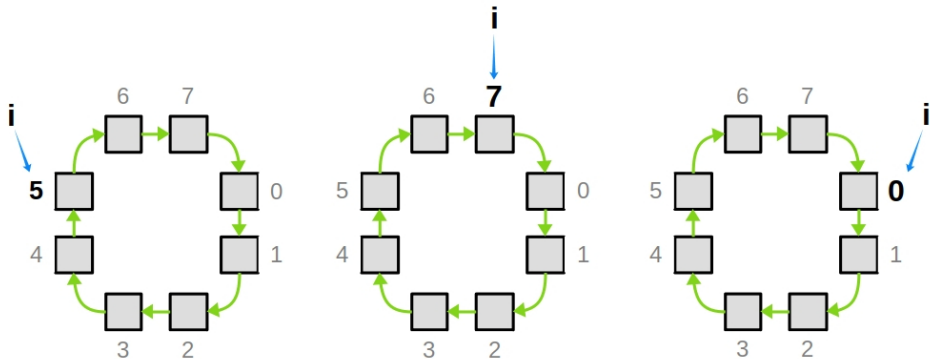
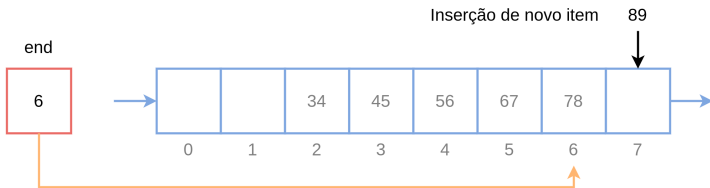


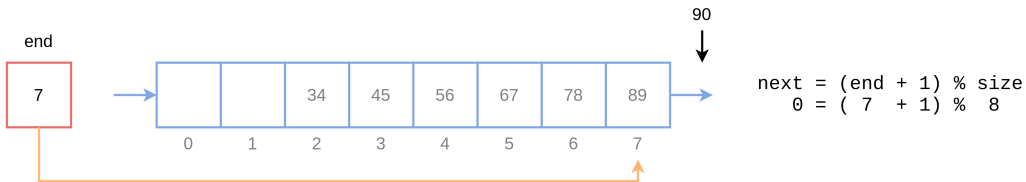
Figura 3: Simulação de vetor circular: após a última posição, o índice volta à primeira.

● Exemplo:

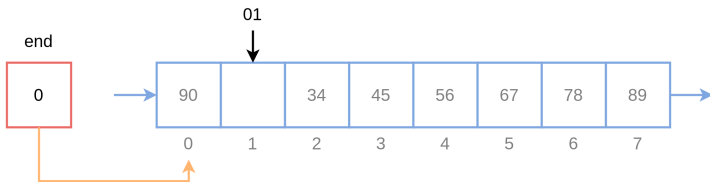


$$\begin{aligned} \text{next} &= (\text{end} + 1) \% \text{size} \\ 7 &= (6 + 1) \% 8 \end{aligned}$$

● Exemplo:

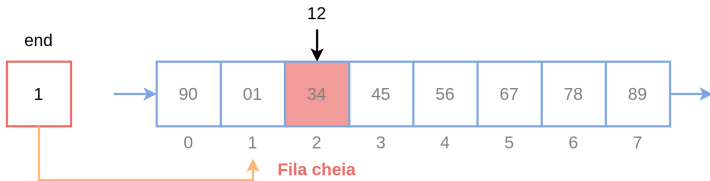


● Exemplo:



```
next = (end + 1) % size  
1 = ( 0 + 1) % 8
```


● Exemplo:



$$\begin{aligned} \text{next} &= (\text{end} + 1) \% \text{size} \\ 2 &= (1 + 1) \% 8 \end{aligned}$$

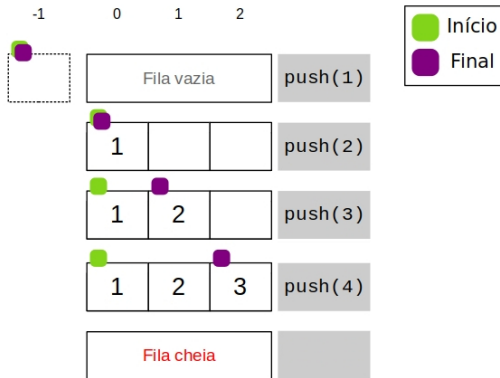
- Adicionaremos ao arquivo de implementação da fila **Queue.c** uma função auxiliar que retorna os índices de forma modular de acordo com sua capacidade.

```
1  int Queue_next(Queue *q, int index) {  
2      if (queue != NULL && index >= 0)  
3          return (index + 1) % queue->size;  
4      else  
5          return 0;  
6  }
```

- Quando um índice indicando a última posição desse vetor for avançado, ele voltará a indicar a primeira posição.
- Posições desocupadas por itens removidos da fila podem ser reusadas para a inserção de novos itens.

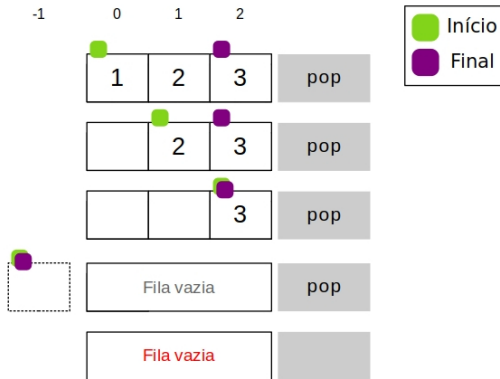
● Inserção em fila

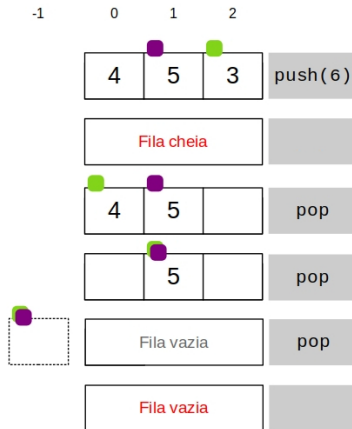
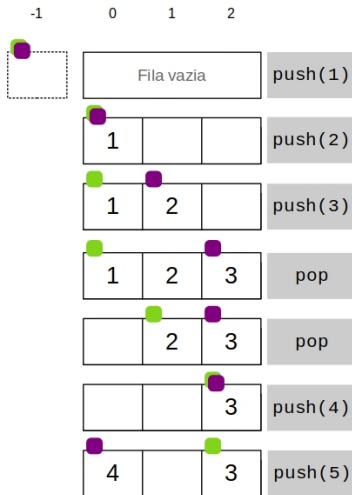
- Para inserir um item numa fila, primeiro temos que verificar se há espaço.
- Caso a fila esteja cheia, a inserção é abortada.
- Do contrário, o item deve ser inserido no final da fila.



Remoção em fila

- Para remover um item de uma fila, primeiro essa função verifica se a fila está vazia.
- Caso a fila esteja vazia, a remoção é abortada.
- Do contrário, o item deve ser removido no início da fila.





Exercício 1

O programa do slide 10 não reconhece “Amor a Roma” como uma cadeia palíndroma. Use a função `toupper()`, declarada em `ctype.h`, para resolver esse problema (essa função converte uma letra minúscula em maiúscula).

Exercício 2

O programa a seguir simula o compartilhamento de uma CPU entre vários processos que aguardam numa fila para serem executados. Enquanto a fila não fica vazia, o primeiro processo na fila pode usar a CPU por certo período de tempo. Se nesse período o processo termina, ele é removido da fila; senão, ele volta para o final dela e o próximo processo na fila passa a usar a CPU. Nessa fila, um processo precisa de t unidades de tempo para concluir sua execução. Analise o programa `processQueue.c` e indique a ordem de conclusão dos processos.

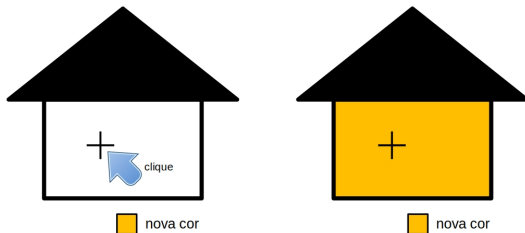
Código 2: processQueue.c

```
1  #include <stdio.h>
2  #include "Queue.h"
3  #define TIME 3
4  #define SIZE 4
5
6  int main() {
7      int p, id, time, procs[SIZE] = {17,25,39,46};
8      Queue *q = Queue_alloc(SIZE);
9      int i;
10
11     for (i = 0; i < SIZE; i++)
12         Queue_push(q, procs[i]);
```



```
14
15     while (!Queue_isEmpty(q)) {
16         p = Queue_pop(q);
17         id = p / 10;
18         time = p % 10
19         time -= TIME;
20
21         if (time > TIME) {
22             p = id*10 + time;
23             Queue_push(q, p);
24         }
25         else
26             printf("Processso %d concluído\n", id);
27     }
28     Queue_free(q);
29     return 0;
30 }
```

- Usaremos o tipo Fila para criar um programa que colore regiões de uma imagem com uma nova cor.
- Essa operação é comum em programas de desenho interativo, como ilustrado na Figura



- **Algoritmos de coloração de imagem**

- Há dois tipos de algoritmos de coloração de imagem:

1. Coloração limitada por área (*flood-fill*) considera regiões monocromáticas limitadas por bordas policromáticas.

Dado um pixel **p** de cor **a**, uma nova cor **n** é propagada para todo pixel de cor **a** na região de **p**.

2. Coloração limitada por borda (*boundary-fill*) considera regiões policromáticas limitadas por bordas monocromáticas.

Dado um pixel **p** e uma cor de borda **b**, uma nova cor **n** é propagada para todo pixel de cor distinta de **n** e de **b** na região de **p**.

- A figura abaixo mostra a diferença entre os resultados obtidos com esses dois tipos de algoritmos.

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
2	0	0	0	1	1	1	0	0	0
3	0	0	1	1	1	1	1	0	0
4	0	1	1	1	1	1	1	1	0
5	0	0	2	0	0	0	2	0	0
6	0	0	2	0	0	0	2	0	0
7	0	0	2	2	2	2	2	0	0
8	0	0	0	0	0	0	0	0	0

(a) Imagem original

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	3	0	0	0	0
2	0	0	0	3	3	3	0	0	0
3	0	0	3	3	3	3	3	0	0
4	0	3	3	3	3	3	3	3	0
5	0	0	2	0	0	0	2	0	0
6	0	0	2	0	0	0	2	0	0
7	0	0	2	2	2	2	2	0	0
8	0	0	0	0	0	0	0	0	0

(b) Limitado por área de cor 1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	3	0	0	0	0
2	0	0	0	3	3	3	0	0	0
3	0	0	3	3	3	3	3	0	0
4	0	3	3	3	3	3	3	3	0
5	0	0	3	0	0	0	3	0	0
6	0	0	3	0	0	0	3	0	0
7	0	0	3	3	3	3	3	0	0
8	0	0	0	0	0	0	0	0	0

(c) Limitado por área de cor 0

- **Coloração limitada por área**

- Os vizinhos de um pixel em uma imagem são aqueles que se encontram acima, à direita, abaixo ou à esquerda dele, como ilustrado na Figura.

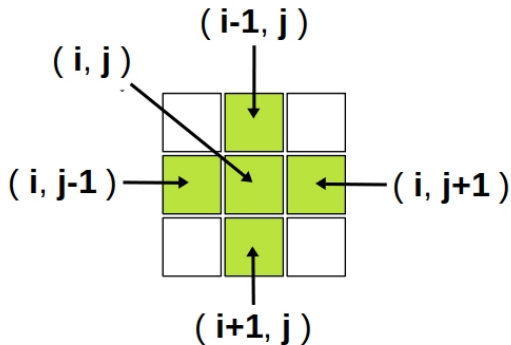


Figura 4: Vizinhança de um pixel numa posição (i, j) de uma imagem.

- Sejam p e q dois pixels de mesma cor. Então, q está na região de p se ele é vizinho de p ou de algum outro pixel que está na região de p .
- Com base nessa definição, dados um pixel p e uma nova cor n , a coloração da região de p é feita do seguinte modo:
 1. Crie uma fila vazia F .
 2. Obtenha cor atual a de p .
 3. Mude a cor de p para n .
 4. Enfileire p em F .
 5. Enquanto a fila F não estiver vazia, faça:
 - 5.1 Desenfileire um pixel p de F .
 - 5.2 Para cada vizinho q de p , que tenha a cor a , faça:
 - 5.2.1 Mude a cor de q para n .
 - 5.2.2 Enfileire q em F .

- A finalidade da fila F nesse processo é manter os pixels já coloridos, até que seus vizinhos possam ser inspecionados e, eventualmente, também coloridos.
- Quando F fica vazia, temos certeza de que todos os pixels de cor a , acessíveis a partir de p , foram coloridos com a nova cor n .

● Representação e exibição de imagem

- A imagem a ser manipulada pelo algoritmo de coloração de regiões será representada por uma matriz de números inteiros positivos.

```
1  #define DIM 9
2
3  int img[DIM][DIM] = {{0, 0, 0, 0, 0, 0, 0, 0, 0},
4                        {0, 0, 0, 0, 1, 0, 0, 0, 0},
5                        {0, 0, 0, 1, 1, 1, 0, 0, 0},
6                        {0, 0, 1, 1, 1, 1, 1, 0, 0},
7                        {0, 1, 1, 1, 1, 1, 1, 1, 0},
8                        {0, 0, 2, 0, 0, 0, 2, 0, 0},
9                        {0, 0, 2, 0, 0, 0, 2, 0, 0},
10                       {0, 0, 2, 2, 2, 2, 2, 0, 0},
11                       {0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

- A função `show()`, exibe no vídeo a imagem representada pela matriz `img`.

```
1 void show(int img[DIM][DIM]) {  
2     int i, j;  
3     for (i = 0; i < DIM; i++) {  
4         for (j = 0; j < DIM; j++)  
5             color(img[i][j]);  
6         printf("\n");  
7     }  
8 }
```

- A função `color()` é usada para colorir a posição que representa o pixel (i, j) .

```
1 void color(int color) {  
2     switch (color) {  
3         case 0 : printf("\033[107m  \033[m"); break;  
4         case 1 : printf("\033[102m  \033[m"); break;  
5         case 2 : printf("\033[100m  \033[m"); break;  
6         case 3 : printf("\033[103m  \033[m"); break;  
7         case 4 : printf("\033[106m  \033[m"); break;  
8     }  
9 }
```

Referência externa: https://en.wikipedia.org/wiki/ANSI_escape_code

● Coloração limitada por área

```
1  int pixel(int i, int j) {
2      return i * DIM + j;
3  }
4
5  void coloringArea(int img[DIM][DIM], int i, int j, int n) {
6      Queue *q = Queue_alloc(DIM*DIM);
7      int p, a = img[i][j];
8
9      img[i][j] = n;
10     Queue_push(q, pixel(i, j));
11
12     while (!Queue_isEmpty(q) && a != n) {
13         p = Queue_pop(q);
14         i = p / DIM;
15         j = p % DIM;
```

```
18     if (img[i][j+1] == a) {
19         img[i][j+1] = n;
20         Queue_push(q, pixel(i, j+1));
21     }
22     if (img[i+1][j] == a) {
23         img[i+1][j] = n;
24         Queue_push(q, pixel(i+1, j));
25     }
26     if (i > 0)
27         if (img[i-1][j] == a) {
28             img[i-1][j] = n;
29             Queue_push(q, pixel(i-1, j));
30         }
31     if (j > 0)
32         if (img[i][j-1] == a) {
33             img[i][j-1] = n;
34             Queue_push(q, pixel(i, j-1));
35         }
36     }
37     Queue_free(q);
38 }
```

Exercício 3

Crie `void initImage(int I[9][9], char *s)` uma função que inicia uma matriz `I` de 9 linhas por 9 colunas com dados lidos de um arquivo de texto, cujo nome é dado pela cadeia `s`. Por exemplo, para iniciar uma matriz com os dados do arquivo `image.txt`, basta fazer a chamada `initImage(I, "image.txt")`. Usando essa função, crie um programa que leia uma imagem de um arquivo do usuário.

Código 3: Arquivo image.txt

```
2 2 2 2 3 2 2 2 2
2 2 2 3 3 3 2 2 2
2 2 3 3 1 3 3 2 2
2 3 3 1 1 1 3 3 2
3 3 1 1 4 1 1 3 3
2 3 3 1 1 1 3 3 2
2 2 3 3 1 3 3 2 2
2 2 2 3 3 3 2 2 2
2 2 2 2 3 2 2 2 2
```

Exercício 4

Usando **coloração limitada por borda**, crie uma versão da função `coloringImage` apresentada no slide 36 e faça um programa para testá-la.