5

Representando filas na linguagem C

As filas também podem ser representadas por um vetor. O vetor é utilizado para armazenar os elementos da fila juntamente com duas variáveis, *inic* e *fim*, que representam, respectivamente, as posições dentro do vetor do primeiro e do último elementos dessa fila.

Uma estrutura pode ser usada para representar uma fila da seguinte forma:

```
#define MAXFILA 100
struct queue {
      int item[MAXFILA];
      int inic, fim;
};
```

A declaração da fila *q* pode ser realizada assim:

```
struct queue q;
```

Para simplificar a explicação da apresentação inicial da implementação da operação de inserção de um novo elemento em uma fila, vamos ignorar momentaneamente a possibilidade de *overflow* e *underflow*. Com essas duas premissas, a seguir está uma sugestão de implementação para a operação *insFila(q,x)*, que faz a inserção de elementos na fila *q*. Observe:

```
q.item [++ q.fim] = x;
```

Ainda, considerando as duas premissas anteriores, a operação *remFila(q)*, que faz a remoção de elementos de uma fila q, pode ser implementada como a seguir. Observe:

```
x= q.item [q.inic ++];
```

A chamada da operação *remFila(q)* pode ser realizada da seguinte forma:

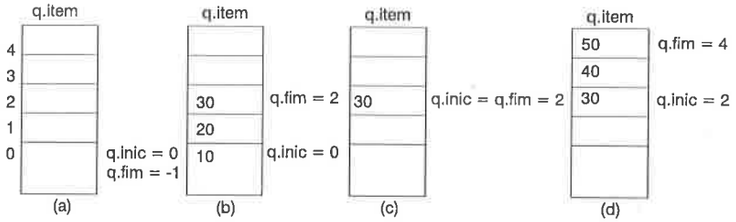
```
x=remFila(q);
```

No momento em que a fila é criada, *q.fim* é definido como -1 e *q.inic* como 0. A fila está vazia sempre que *q.fim* < *q.inic*. O número de elementos nessa fila é sempre igual ao valor de *q.fim - q.inic + 1*. Com essas duas últimas considerações, notamos que a fila, ao ser inicializada como vazia tem zero elementos (-1+0+1=0).

Essa representação inicial de filas com vetores é uma forma rápida de materializar as filas como uma estrutura de dados para que possamos utilizar as filas nas soluções de problemas. Realizando uma análise na forma que a estrutura da fila foi definida e nas implementações sugeridas para as operações de inserção e remoção, podemos observar que à medida que se inserem elementos novos na fila *q*, *q.fim* é incrementado em uma posição. Na remoção é a vez de *q.inic* ser incrementado em uma posição. Analisando este processo, notamos que o valor de *q.inic* caminha em direção ao valor de *q.fim*, até o momento que *q.fim* torna-se menor do que *q.inic*. Neste caso, é sinalizado que a fila *q* está vazia. Lembre-se da condição *q.fim* < *q.inic* que define a fila como vazia. Agora, vamos apresentar uma série de situações que mostrará que essa representação simplista das filas pode conduzir a observação de alguns inconvenientes.

O primeiro inconveniente a ser considerado mediante essa representação simplista das filas, é a situação um tanto quanto absurda de obtermos uma fila vazia, porém sem a possibilidade de inserirmos novos elementos. Observe:

Figura 5.1
Operações sobre uma fila.



A ilustração da Figura 5.1 mostra quatro momentos diferentes de uma mesma fila, a qual pode conter, no máximo, cinco elementos numéricos inteiros.

No momento (a) da Figura 5.1, a fila está vazia, e isso é indicado por *q.inic = 0* e *q.fim = -1*. Nesse momento, a quantidade de elementos é dada por *q.fim - q.inic + 1*, ou seja, *-1 - 0 + 1 = 0*.

Em (b) também dessa figura, houve a inclusão de três elementos numéricos, que são 10, 20 e 30, resultando na seguinte quantidade de elementos: 2 - 0 + 1 = 3. No momento (b), o primeiro elemento da fila é indexado pelo valor de *q.inic*, o que acaba resultando em 10. O último elemento da fila no momento (b) está indexado pelo valor de *q.fim*, resultando, dessa forma, em 30.

Avançando para o momento (c) dessa Figura, podemos perceber que houve a remoção dos elementos 10 e 20. Isso é indicado porque o valor de *q.fim* é igual ao valor de *q.inic*. Isso indica também que 30 é ao mesmo tempo o primeiro e o último elemento da fila. Na situação (c), a quantidade de elementos é dada por 2- 2 + 1 = 1.

Finalmente, chegamos ao momento (d) da Figura 5.1, onde *q.inic = 2* e *q.fim = 4*. Esses valores indicam que existem 4 - 2 + 1 = 3 elementos na fila. Se houver a necessidade de adicionarmos mais valores no vetor que representa a fila, percebemos que isso não será possível, pois o elemento 50 já está ocupando a última posição do vetor. Observamos, contudo, que existem duas posições livres (0 e 1) no vetor que representa a fila. Além disso, já foi dito que uma fila vazia ocorre quando *q.fim* < *q.inic*. Atingimos a situação absurda de termos espaços livres em um vetor que representa uma fila, mas, devido as considerações simplistas da implementação da fila, não é possível realizarmos a inclusão de outro elemento na fila.

O impasse descrito anteriormente pode ser solucionado, modificando-se a operação *remFila(q)* de tal forma que, ao ser eliminado um item, realiza-se uma movimentação da fila como um todo, em direção ao início do vetor. Se ignorarmos novamente a possibilidade de *underflow*, a implementação da operação *remFila(q)* pode apresentar-se como o que é exibido a seguir:

```
x = q.item [ 0 ];
for (k = 0); k < q.fim; k++)
      q.item[ k ] = q.item[ k + 1 ];
q.fim - -;
```

O campo *inic* não é necessário, porque o elemento na posição 0 do vetor está sempre no início da fila. A fila vazia é representada por *q.fim* igual a -1.

Podemos ver claramente que existiria um grande esforço computacional para a movimentação de 500, 1.000, 10.000, 1.000.000 elementos. Essa solução escolhida parece ser bastante ineficiente. A seguir, é apresentada a solução de nome fila circular, que elimina a situação absurda anteriormente descrita. Além do mais, essa situação será contornada sem haver a necessidade de movimentar os elementos presentes na fila circular.

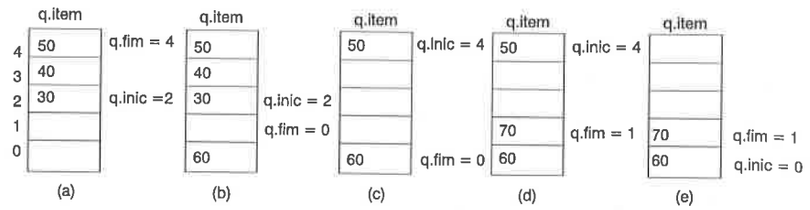
Fila circular

A fila circular apresenta-se como uma solução muito elegante para a resolução das questões anteriormente discutidas. A idéia é armazenar os elementos na fila como se esta fosse um círculo.

Para compreendermos essa solução, é necessário imaginar que o primeiro elemento do vetor vem logo depois do seu último. Isso significa que, se o último elemento estiver ocupado, um novo valor poderá ser inserido depois dele, nesse caso, o primeiro elemento do vetor desde que esteja obviamente vazio. Podemos notar que um elemento novo não será incluído em uma fila circular somente se não houver, de fato, espaço nessa fila.

A ilustração a seguir pode ajudar no entendimento do mecanismo de funcionamento da fila circular. Observe:

Figura 5.2
Ilustração do mecanismo de funcionamento da fila circular.



Acompanhando a evolução dos cinco momentos da fila circular presentes na Figura 5.2, podemos perceber que a identificação do primeiro e do último elementos presentes na fila depende de uma análise dos valores de *q.inic* e *q.fim*. Por exemplo, no momento (b) da Figura 5.2, o primeiro elemento da fila é 30 e o último é 60. Em (d) dessa mesma figura, o primeiro elemento é 50 e o último é 70. Notamos que o problema discutido anteriormente desapareceu, pois enquanto houver de fato espaço no vetor que representa a fila circular, a inclusão será sempre possível.

Entretanto, considerando o momento (d) presente na Figura 5.2, podemos ver que a expressão *q.fim* < *q.inic* (1 < 4) é verdadeira. Poderíamos, dessa forma, concluir que a fila está vazia, quando de fato não está. Devido à implementação circular da fila, não podemos mais simplesmente comparar se *q.fim* é menor do que *q.inic*, para verificarmos se uma pilha está vazia. Então surge uma outra questão: como determinar se a fila está vazia?

A resposta para a questão colocada é o estabelecimento de uma convenção em relação ao valor de *q.inic*. Assume-se que *q.inic* é o índice do vetor que é exatamente anterior ao primeiro elemento da fila, em vez de ser o índice do próprio primeiro elemento. Como *q.fim* é o índice do último elemento da fila, a convenção adotada faz com que uma fila seja vazia quando observarmos a condição *q.inic* = *q.fim*.

Para que essa convenção funcione, uma fila de inteiros passa a ser declarada e iniciada conforme o próximo exemplo:

```
#define MAXFILA 100
struct queue {
      int item [ MAXFILA];
      int inic, fim;
};
struct queue q;
q.inic = q.fim = MAXFILA - 1;
```


Com a convenção adotada para a implementação de fila circular, é necessário que *q.inic* e *q.fim* sejam inicializados com o último índice do vetor, em vez de -1 e 0. A razão dessa mudança é que, com essa representação, o último elemento do vetor precede imediatamente o primeiro dentro da fila. Fazendo-se essa inicialização, *q.fim* será igual a *q.inic*; portanto, a fila estará inicialmente vazia. Apresentamos a seguir uma sugestão da rotina *inicFila( )*, que inicializa uma fila sob essa nova representação.

```
void inicFila(struct queue *pq)
{
      pq→inic = MAXFILA -1;
      pq→fim  = MAXFILA -1;
}
```

Na fila circular, a função *filaVazia( )*, que indica se uma fila está ou não vazia, pode ser codificada como:

```
int filaVazia(struct queue *pq)
{
      if (pq→inic==pq→fim)
            return (1); /*1 é verdadeiro*/
      else
            return (0);/* 0 é falso*/
} /* fim filaVazia*/
```

Então, o uso da função *filaVazia( )* para identificar se uma fila circular está vazia pode ser codificada como:

```
if (filaVazia((&q)==1)
      /* tratamento adequado quando a fila está vazia       */
else
      /* tratamento adequado quando a fila não está vazia   */
```

A operação *remFila(&q)* para remover um elemento da fila circular pode ser codificada conforme é ilustrado a seguir. As mesmas observações já realizadas durante a implementação dos algoritmos das pilhas em relação ao uso de *exit ( )* valem também para a implementação dos algoritmos das filas. Lembre-se de que é necessário um tratamento mais adequado para que o programa sinalize ou se recupere do erro causado pela possibilidade de ocorrência de *overflow* ou de *underflow* nas filas. Observe:

```
int remFila(struct queue *pq)
{
      if (filaVazia(pq)) {
            printf("Underflow na fila!\n");
            exit ( 1);
      }
```

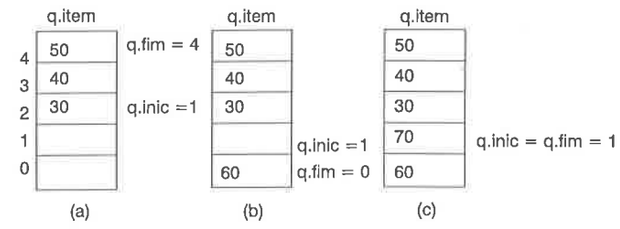
```
        if (pq → inic = = MAXFILA - 1)
             pq → inic = 0;
        else
             ( pq →inic)++;
        return  ( pq → item [ pq → inic ] );
} /* fim  remove */
```

O motivo pelo qual não colocamos o operador de endereço & em *pq* na chamada de *filaVazia(pq)*, é que *pq* já é um ponteiro para a fila *q*. Observe que na implementação sugerida para *remFila( )*, foi considerada a prevenção da ocorrência de um *underflow* que pode ocorrer ao tentarmos remover um elemento de uma fila vazia.

A operação de inserção nas filas

A operação de inclusão de um novo elemento em uma fila circular representada por um vetor deve envolver o teste de estouro (*overflow*), que ocorre quando o vetor inteiro é ocupado por itens da fila, fazendo uma tentativa de inserção de outro elemento na fila. Observe a seguir:

Figura 5.3
Ilustração da operação da inserção na fila.



Acompanhando os três momentos ((a), (b) e (c)) da fila na ilustração da Figura 5.3, percebemos que a determinação de *overflow* no momento (c), quando ocorre a inclusão de um novo elemento na fila circular, é confundido com a ocorrência de um *underflow*, pois, no momento (c), *q.inic* é igual a *q.fim*. Para entendermos melhor essa situação, devemos verificar com cuidado a definição da função *filaVazia( )*. Podemos notar que uma fila estará vazia no momento *q.inic* tornar-se igual a *q.fim*. Entretanto, no momento (c) da Figura 5.3, a fila está cheia e não vazia. Este novo impasse tornou insatisfatória a implementação da fila circular. É necessário acharmos uma solução para contornar essa situação na qual o *overflow* pode ser confundido com o *underflow*.

Uma solução que pode ser adotada é sacrificar uma posição do vetor para diferenciarmos a ocorrência do *overflow* do *underflow*. Isso significa permitir que o vetor aumente somente até um elemento abaixo do tamanho máximo do vetor. Por exemplo, se uma fila utilizar um vetor

de cem elementos, essa fila poderá conter, no máximo, 99 elementos. A tentativa de inserir o centésimo elemento na fila será detectada como a ocorrência de um estouro (*overflow*), e o programa deverá tomar as ações adequadas. Com essa solução, a condição assumida de *q.inic = q.fim* para a verificação do *underflow* permanece intacta. Agora, a implementação da fila circular sob a forma de vetor ficou completa e pode ser utilizada na construção de algoritmos que visem a solução de problemas.

Nesta representação final, poderemos sempre inserir elementos em uma fila circular, sem a necessidade de movimentar os elementos presentes na mesma, desde que realmente haja espaço na fila circular.

Em seguida apresentamos uma sugestão de implementação da rotina *insFila (q,x)* que realiza a inserção de um elemento em uma fila circular. Observe:

```
void insFila( struct queue *pq, int x)
{
     /*realiza a movimentação para abrir espaço para novo elemento */
     if (pq → fim = = MAXFILA - 1)
          pq → fim = 0;
     else
          (pq → fim )++;
     /* verifica ocorrência de estouro */
     if (pq → fim = = pq →inic) {
          printf ("Ocorreu overflow na fila!\n");
          exit ( 1 );
     }
     pq → item[ pq → fim] = x;
     return;
}
```

A grande diferença entre as funções *insFila( )t* e *remFila( )* é que o teste de estouro em *insFila( )* ocorre somente depois que *p→ fim* é incrementado em uma unidade. Já o teste de underflow em *remFila( )* é realizado assim que a rotina é inserida; só então *pq→ inic* é incrementado em uma unidade.

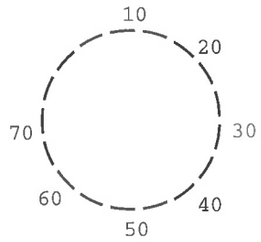
Após a apresentação das principais operações que atuam sobre as filas e suas respectivas implementações na linguagem C, apresentamos a seguir um programa muito interessante que permite, de forma didática, que o leitor visualize na tela do computador o funcionamento do mecanismo de inserção e de remoção de elementos em uma fila circular. Recomendamos que o leitor faça um estudo da implementação sugerida. Note o uso da biblioteca *queue.h* contendo as operações sobre as filas. O conteúdo de *queue.h* é apresentado logo após o programa. O programa em execução deverá exibir a tela abaixo. Observe:

*** Exemplo de fila circular***

*** Menu ***

1 - Incluir elemento
2 - Remover elemento
3 - Sair
Digite sua opção : 1
Elemento : 80

Primeiro elemento : 0
Último elemento : 6
Quantidade de elementos : 7
Status : Overflow




```
/*********************************************************************/
/* Programa que mostra o funcionamento de uma fila circular e faz  */
/* a utilizacao das funcoes de manipulacao de filas.               */
/*********************************************************************/

#include "queue.h"

/*Prototipos do desenho da fila circular*/
void Draw_Menu();
void Draw_Queue(struct queue *pq);

/*Variavel que indica o status da fila*/
int status=0;


main()
     {
     int elem, op;
     struct queue q;

     /*Inicializa a fila*/
     inicFila(&q);
     op=0;
     /*Loop principal: para sair do programa devera ser escolhida a
opcao 3*/
     while(op!=3)
          {
          op=0;

          /*Desenhando o Menu Principal*/
          Draw_Menu();

          /*Desenhando a lista*/
          Draw_Queue(&q);
```



```
          /*Recebendo a opcao ate' que seja fornecida uma valida*/
          while(op<1 || op>3)
               {
               gotoxy(26,11);
               scanf("%i",&op);
               }

          /*Menu*/
          switch(op)
               {
               case 1:
                    if(filaCheia(&q))
                         {
                         status=2;
                         }
                    else
                         {
                         /*Recebendo o valor a inserir na lista ...*/
                         gotoxy(8,12);
                         printf("Digite o elemento: ");
                         gotoxy(27,12);
                         scanf("%i",&elem);
                         insFila(&q,elem);
                         status=0;
                         }
               break;
               case 2:
                    if(filaVazia(&q))
                         {
                         status=1;
                         }
                    else
                         {
                         remFila(&q);
                         status=0;
                         }
                    break;
                    }
               }
          }

/*Função que desenha o Menu Principal*/
void Draw_Menu()
     {
     /*Formatando a tela*/
     clrscr();

     /*Titulo e Menu*/
     gotoxy(27,3);
     printf("*** Exemplo de Fila circular***");
     gotoxy(12,5);
```

```
        printf("*** Menu ***");
        gotoxy(8,7);
        printf("1 - Incluir elemento");
        gotoxy(8,8);
        printf("2 - Remover elemento");
        gotoxy(8,9);
        printf("3 - Sair");
        gotoxy(8,11);
        printf("Digite sua opcao: ");
        }

/*Desenha a Lista*/
void Draw_Queue(struct queue *pq)
        {
        int i=0, count=0, first=MAXFILA-1;

        /*Loop para desenhar os numeros*/
        for(i=pq->inic+1;i!=pq->fim+1;i++)
                {
                if(i==MAXFILA)
                        i=0;

                if(!count)
                        first=i;

                switch(i)
                        {
                        case 0:
                                gotoxy(55,7);
                        break;
                        case 1:
                                gotoxy(64,9);
                        break;
                        case 2:
                                gotoxy(68,12);
                        break;
                        case 3:
                                gotoxy(64,15);
                        break;
                        case 4:
                                gotoxy(55,17);
                        break;
                        case 5:
                                gotoxy(46,15);
                        break;
                        case 6:
                                gotoxy(42,12);
                        break;
                        case 7:
                                gotoxy(46,9);
                        break;
```



```
                }
                printf("%i",pq->item[i]);
                count++;
                }

        gotoxy(8,18);
        printf("Primeiro elemento.....: %i",first);
        gotoxy(8,19);
        printf("ultimo elemento.....: %i",pq->fim);
        gotoxy(8,20);
        printf("Quantidade de elementos.: %i",count);
        gotoxy(8,21);
        printf("Status.....: ");

        switch(status)
                {
                case 0:
                        gotoxy(34,21);
                        printf("Normal");
                break;
                case 1:
                        gotoxy(34,21);
                        printf("Underflow");
                break;
                case 2:
                        gotoxy(34,21);
                        printf("Overflow");
                break;
                }
        }
```

Logo a seguir, são apresentados os protótipos e as definições das operações que atuam sobre uma fila circular e que podem ser utilizados para o desenvolvimento da biblioteca *queue.h*, criando, dessa forma, facilidades no uso das filas nos programas.

```
/*********************************************/
/*Protótipos das funções que atuam sobre filas*/
/*********************************************/
/*Estrutura da fila*/
#define MAXFILA 8
struct queue
        {
        int item[MAXFILA];
        int inic, fim;
        };

/*Inicializa a estrutura da fila*/
void inicFila(struct queue *pq);
```

```
/*Insere o elemento x na fila*/
void insFila(struct queue *pq,int x);

/*Remove um elemento da fila*/
int remFila(struct queue *pq);

/*Verifica se a fila esta vazia*/
int filaVazia(struct queue *pq);

/*Verifica se a fila esta cheia*/
int filaCheia(struct queue *pq);

/****************************/
/* Definições das operações */
/****************************/

/*Inicializa a estrutura da fila*/
void inicFila(struct queue *pq)
     {
     pq->inic=MAXFILA-1;
     pq->fim=MAXFILA-1;
     }

/*Insere o elemento x na fila*/
void insFila(struct queue *pq,int x)
     {
     if(pq->fim==MAXFILA-1)
          pq->fim=0;
     else
          (pq->fim)++;

     if(pq->fim==pq->inic)
          {
          printf("Ocorreu overflow na fila!\n");
          exit(1);
          }

     pq->item[pq->fim]=x;
     }

/*Remove um elemento da fila*/
int remFila(struct queue *pq)
     {
     if(filaVazia(pq)==1)
          {
          printf("Ocorreu underflow na fila!\n");
          exit(1);
          }
```



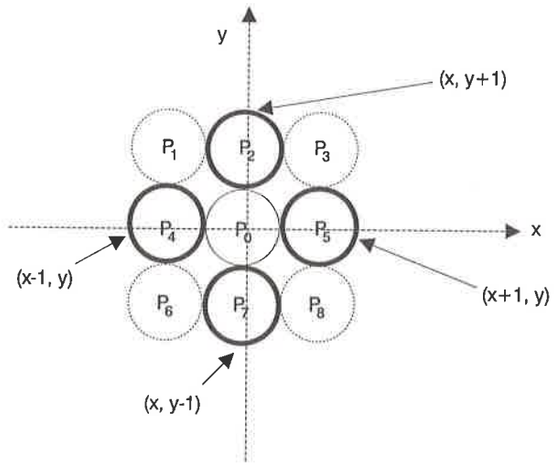
```
     if(pq->inic==MAXFILA-1)
          pq->inic=0;
     else
          (pq->inic)++;

     return(pq->item[pq->inic]);
     }

/*Verifica se a fila esta vazia*/
int filaVazia(struct queue *pq)
     {
          if (pq->inic==pq->fim)
               return (1) ; /*verdadeiro*/
          else
               return (0); /*falso*/
     }

/*Verifica se a fila esta cheia*/
int filaCheia(struct queue *pq)
     {
     if(pq->fim==MAXFILA-1)
          return(pq->inic==0);
     else
          return(pq->inic==((pq->fim)+1));
     }
```

Filas em algoritmos de coloração

O uso de filas pode colaborar para o desenvolvimento de algoritmos com o intuito de realizar a coloração de regiões de desenho representadas sob a forma de matrizes de pontos.

Podemos definir uma região de um desenho de nome R como um conjunto de pontos conectados entre si apresentando a mesma cor (Figura 5.4). Consideramos que dois pontos, P_i e P_j, são conectados entre si se, e somente se, ao incrementarmos ou decrementarmos a abscissa (ou a ordenada) do ponto de origem P_i, chegarmos ao ponto P_j. A ilustração da Figura 5.4 pode ajudá-lo no entendimento quando dois pontos estão ou não conectados.

Figura 5.4
Representação de uma região de desenho.



Podemos observar que quatro pontos presentes no desenho da Figura 5.4, P_1 (x-1, y+1), P_3 (x+1, y+1), P_6 (x-1, y-1) e P_8 (x+1, y-1), não estão conectados a P_0 (x,y), porque existe a necessidade de variarmos x e y ao mesmo tempo, caso queiramos alcançar P_0. Isso não ocorre com os pontos P_2, P_4, P_5 e P_7. Faça você mesmo a verificação.

Podemos especificar um algoritmo para colorir a região R da seguinte forma:

Passo 1
Determinar um ponto inicial P_0 de cor C_0 que pertença de fato à região R.

Passo 2
Determinar a nova cor C_n para a região R.

Passo 3
Empilhar o ponto inicial P_0 em uma fila q, inicialmente vazia.

Passo 4
Enquanto a fila *q* não esvaziar:
- Desempilhar um ponto P da fila *q*;
- Empilhar em *q* todos os pontos conectados a P cuja cor seja C_0;
- Mudar a cor de P para C_n.

Uma simulação do algoritmo anteriormente apresentado pode ajudar no entendimento. Suponha a existência de uma tabela de cor na forma C1= branco, C2= cinza e C3=preto. Considere que seja necessário fornecer o ponto inicial e a cor. Assuma que as informações são P_0(3,3) e C_3 = preto, respectivamente, para o ponto inicial e para a nova cor desejada para esse ponto inicial (Figura 5.5). Como fica a coloração da imagem depois de aplicarmos o

algoritmo de coloração apresentado? Na Figura 5.6 podemos acompanhar visualmente a evolução do processo de coloração, considerando o algoritmo anteriormente apresentado, o ponto inicial como P_0(3,3) e a nova cor que será utilizada para colorir a região de desenho, no caso C_3 = preta. Uma fila *q* é apresentada para acompanharmos a entrada e a saída dos pontos que aguardam a vez para receber a coloração desejada.

Figura 5.5
O ponto P_0 inicial da região de desenho.

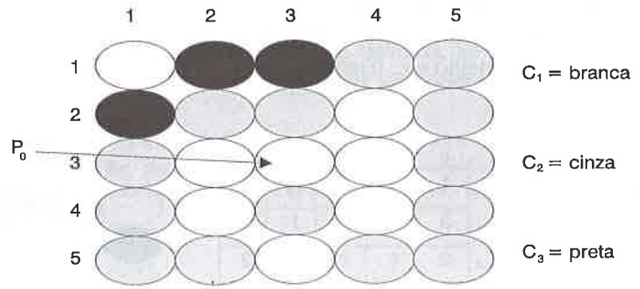
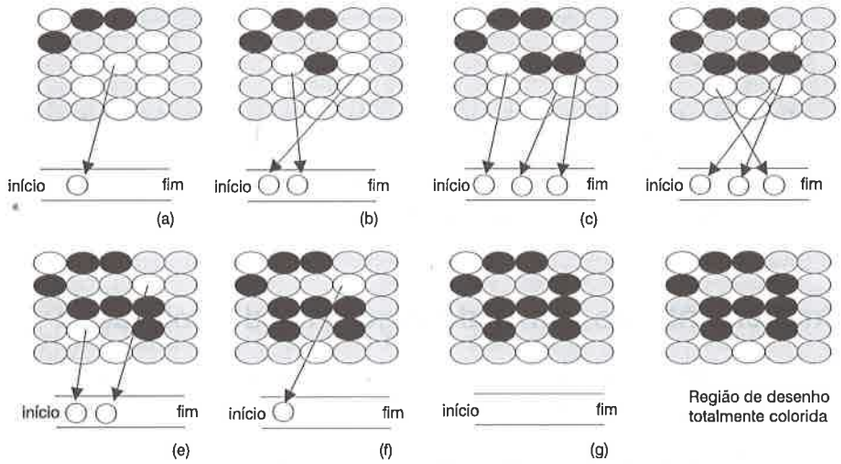


Figura 5.6
Evolução do processo de coloração com filas.



Durante a evolução do algoritmo, notamos que alguns pontos do desenho, não pertencentes a mesma região do ponto P_0, permanecem na cor branca. É possível verificar também que um mesmo ponto entra mais de uma vez na fila para ser colorido, mas podemos considerar esse fato irrelevante para o funcionamento do algoritmo.

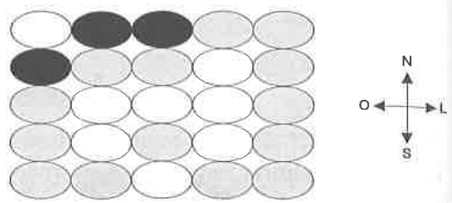
A fila do algoritmo apresentado tem a responsabilidade de ser um local de espera, em que cada ponto aguarda sua vez para ser colorido. No momento em que a fila torna-se vazia, é sinal de que todos os pontos pertencentes a mesma região foram coloridos.

Para implementarmos o algoritmo em uma linguagem de programação é necessário achar uma representação para os pontos luminosos. Uma boa consideração é fazer uma equivalência de um ponto luminoso com um pixel em um monitor de vídeo. Em termos mais práticos, podemos realizar uma abstração de detalhes de *hardware*, representando a região de um desenho por uma matriz bidimensional.

Cada um dos elementos da matriz representa um ponto. Um pixel pode ser discriminado pelas coordenadas da sua posição na matriz. A cor do ponto é armazenada em cada elemento da matriz seguindo uma determinada tabela, como por exemplo: 1 = branco, 2 = cinza, 3 = preto, 4 = vermelho etc. (Figura 5.7).

Figura 5.7
Representação da região de desenho com matriz.

	0	1	2	3	4
0	1	3	3	2	2
1	3	2	2	1	2
2	2	1	1	1	2
3	2	1	2	1	2
4	2	2	1	2	2



Do lado extremo direito da Figura 5.7, podemos ver um desenho representando os quatro pontos cardeais para a realização da orientação no mapeamento da matriz para os pontos da região que será colorida. Essa orientação será usada durante a construção da implementação em C para a realização da coloração de uma região de desenho.

Implementação em C do algoritmo de coloração

Para facilitar a implementação em C do algoritmo de coloração, podemos assumir alguns pressupostos, como por exemplo:

- Fornecimento da matriz que representa o desenho;
- A coordenada do ponto inicial da coloração é fornecida;
- O valor da nova cor também é fornecido;
- A obtenção da cor atual da região C_0 pode ser realizada diretamente da matriz que foi informada.

Existem várias possibilidades de representarmos os pontos que já foram manipulados pelo algoritmo apresentado. Uma possibilidade que podemos considerar a mais fácil é tratarmos os

pontos como coordenadas na forma de dois números inteiros independentes. Essa consideração resulta em duas rotinas auxiliares, *qinsere* e *qremove*, cujo funcionamento é descrito a seguir:

- *qinsere* responde pela colocação de uma coordenada válida x,y na fila.
- *qremove* responde pela remoção de uma coordenada x,y previamente colocada na fila por *qinsere*.

A seguir, podemos encontrar uma implementação do algoritmo de coloração na linguagem C. Observe o uso da biblioteca *'queue.h'* anteriormente definida. Recomendamos que o tamanho do vetor *item [ ]* presente na biblioteca *'queue.h'* seja alterado para 1.000.

```
/*********************************************************************/
/* Coloração de uma região de desenho com filas       */
/*********************************************************************/
#include "queue.h"
#define TAM 5

void colorir(int m[][TAM], int x, int y, int c);
void qinsere(struct queue *pq, int x, int y);
void qremove(struct queue *pq, int *x, int *y);

main()
{
static int
     imagem [TAM][TAM]={{1,3,3,2,2}, /* desenho original */
                        {3,2,2,1,2},
                        {2,1,1,1,2},
                        {2,1,2,1,2},
                        {2,2,0,2,2}};
int
     l,c,p1,p2,p3;
     p1=2; /* linha do ponto inicial P0 (coordenada x)       */
     p2=2; /* coluna do ponto inicial P0 (coordenada y)           */
     p3=3; /* nova cor, 3 = preta                            */
     clrscr();
     printf("Entrada\n");
     for (l=0;l<5;l++){
     for (c=0;c<5;c++)
          printf("%i ",imagem[l][c]);
     printf("\n");
     }
     colorir(imagem, p1,p2,p3);
     printf("\nSaida\n");
     for (l=0;l<5;l++){
     for (c=0;c<5;c++)
          printf("%i ",imagem[l][c]);
     printf("\n");
     }
```

```
        printf("\nDigite <enter> para finalizar . . .");
        getch();
}

void colorir(int m[][TAM], int x, int y, int c1)
{
            struct queue q;
            int c0;
            c0 = m[x][y];                   /* obtem a cor inicial do
ponto P0            */
            inicFila(&q);                   /* inicializa fila          */
            qinsere(&q,x,y);            /* insere o ponto inicial  */
            while ((!filaVazia(&q))==1){    /* enquanto nao esvaziar    */
                  qremove(&q,&x,&y);
                  if (m[x+1][y] ==c0)
                        qinsere(&q, x+1,y);   /* leste                  */
                  if (m[x][y-1] ==c0)
                        qinsere(&q,x,y-1);    /* sul                    */
            if (m[x-1][y] ==c0)
                        qinsere(&q,x-1,y);    /* oeste                  */
            if (m[x][y+1] ==c0)
                        qinsere(&q,x,y+1);    /* norte                  */
                  m[x][y]=c1;               /* altera a cor do ponto    */
        }
}

void qinsere(struct queue *pq, int x,int y)
{
        insFila(pq,x);
        insFila(pq,y);
}

void qremove(struct queue *pq, int *x, int *y)
{
        *x=remFila(pq);
        *y=remFila(pq);
}
```


EXERCÍCIOS DE APROFUNDAMENTO

1. Suponha a existência de uma pilha de inteiros *s* e uma fila de inteiros *q*. Desenhe a ilustração de s e q depois das seguintes operações:

```
Push (s,3)
Push(s,12)
InsFila(q,5)
InsFila(q,8)
x= Pop(s)
Push(s,2)
InsFila(q,x)
Push(s,x)
y= elemTopo(s)
Push(s,y)
```

2. Qual é o conteúdo resultante da fila *q* depois que o seguinte trecho de algoritmo em pseudocódigo é executado e os seguintes valores são inseridos: 5, 7, 12, 4, 0, 4, 6, 8, 67, 34, 23, 5, 0, 44, 33, 22, 6, 0.

```
inicFila (q)
enquanto (não for o final das entradas)  faça
|  leia num
|  se (num ≠ 0)     então
|  |     insFila(q, num)
|  |          Senão
|  |     remFila(q)
|  fim se
fim enquanto
```

3. Dado uma fila de inteiros, escreva um algoritmo que exclua todos os inteiros negativos sem alterar a posição dos outros elementos na fila *q*.

4. Escreva um algoritmo em pseudocódigo que inverta o conteúdo de uma fila.

5. Suponha duas filas chamadas *qpar* e *qimpar* e uma pilha *s*. Considere que as três estruturas são representadas por vetores com, no máximo, 30 elementos. Solicita-se o desenvolvimento de um algoritmo que lê uma seqüência indeterminada de valores inteiros. Assuma que o valor zero finaliza a entrada de dados. Em seguida, deve ser determinado se um certo número lido é par ou ímpar. Se o número for par, devemos colocá-lo na fila *qpar*; caso contrário, na fila *qimpar*. Logo após a entrada do valor zero, alternadamente (começando-se pela fila *qimpar*), devemos retirar um elemento de cada fila até o momento em que ambas tornam-se vazias. Se o elemento retirado de uma das

filas for um valor positivo, devemos colocá-lo na pilha *s*; caso contrário, removemos um elemento da pilha *s*. Quando finalizarmos esse processo, devemos exibir todo o conteúdo da pilha *s*.

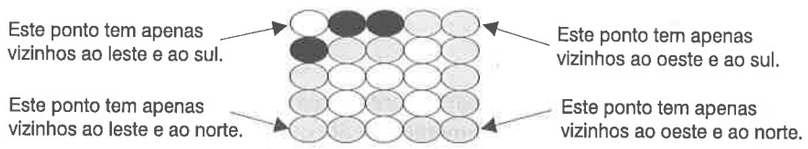
6. Defina o termo fila. Quais operações podem ser realizadas acerca de uma fila?

7. Como é possível a implementação de uma fila circular em um vetor linear?

8. É solicitado que o programa que realiza a coloração de regiões seja completamente finalizado, de forma que o usuário possa definir um desenho qualquer e realizar o processo de coloração da mesma.

9. Desenvolva um programa que use uma pilha em vez de uma fila para a implementação do algoritmo de coloração de regiões. Faça isso de tal forma que uma pessoa possa ver, ao mesmo tempo, a evolução da coloração do algoritmo com a fila e com a pilha. Você pode usar *threads* para mostrar a evolução dos dois algoritmos; entretanto, lembre-se de que, para a comparação tornar-se válida, será necessário o uso do mesmo tempo para os dois *threads*. Levando em consideração o aspecto do espaço, quais seriam as vantagens e as desvantagens das duas possibilidades?

10. O algoritmo e o programa de coloração apresentado no texto não levou em conta a possibilidade de um ponto localizado em um dos quatro extremos da região a ser colorida pudesse ser escolhido como ponto de partida. Se isso for realizado com o algoritmo sugerido no texto, teremos a ocorrência de um erro, pois não será possível fazer a verificação de duas coordenadas, ou seja, o ponto não terá quatro pontos vizinhos, mas sim apenas dois. Solicita-se a modificação adequada do algoritmo e do programa para que essa situação seja tratada de forma adequada. Veja a dica abaixo:



11. Altere o programa de coloração de forma que, ao invés de definir, internamente no programa, o conteúdo da matriz que representa a região de desenho a ser colorida, seja possível o usuário digitar o seu conteúdo. Faça o mesmo para o ponto P_0 e a nova cor.

12. Expanda o programa de coloração já na forma do Exercício 11 para que manipule a região de desenho com cores ao invés de valores numéricos que representam cores.

13. Desenvolver em C a função numFila ( ) que retorna o número de elementos existentes em uma fila circular.

14. Escrever na função primFila ( ) e ultFila ( ) que retorna, respectivamente, o primeiro e o último elemento de uma fila circular.

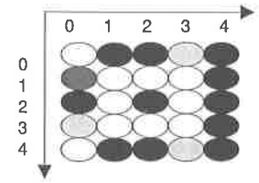
15. O algoritmo em pseudocódigo apresentado abaixo utiliza uma pilha e duas filas. Você acompanhará sua execução, supondo as seguintes entradas: 1, 2, 3, 4, 5, 6, 7, 8, 99. É solicitado que você indique a saída das filas *q1* e *q2* no final da execução do algoritmo.

```
algoritmo exemplo
variáveis
      nr, r , x : inteiro

início
  leia nr
  enquanto nr ≠ 99 faça
       push(s,nr)
       leia nr
  fim enquanto
  enquanto (pilhaVazia ≠ "V") faça
      x← pop(s)
      r← x mod 3
      se r ≠ 0 então
                    x← x*2
                   insFila(q1,x)
             senão
                    x← x*3
                   insFila(q2,x)
      fim se
  fim enquanto
fim.
```

16. Neste capítulo, foi apresentado um algoritmo de coloração na forma narrativa que utilizava as filas. São solicitadas as seguintes tarefas:

   a. Escreva novamente o algoritmo na forma narrativa, utilizando pilhas em vez de filas.

   b. Considere a região do desenho abaixo. Utilizando o algoritmo da Etapa a, mostre a evolução da pilha e da própria região de desenho durante o processo de coloração, supondo que foi escolhido o ponto P_0 (1,1), onde a primeira posição representa o número da linha e a segunda, a coluna. A nova cor do ponto P_0 será preto (1=branco, 2=cinza e 3=preto).



   c. Faça o mesmo que foi solicitado na Etapa b, utilizando dessa vez as filas.

17. Um uso útil das filas refere-se à possibilidade de executar a simulação de um sistema, ou seja, auxiliar na imitação do comportamento de um sistema real. Isso é justificado porque muitas vezes a realização de simulações diretamente sobre os sistemas reais pode ser uma tarefa perigosa ou demorada e até mesmo exigir um alto investimento financeiro para a realização das simulações. As filas são úteis em simulação, porque representam a estrutura perfeita para simular a espera de que muitos objetos necessitam, até chegar sua vez de ser atendido. Podemos citar vários cenários nos quais as filas seriam úteis: simulação de filas em bancos e supermercados, filas de espera de vôo em aeroportos ou ainda nos sistemas operacionais.

    Com o objetivo de demonstrar a possibilidade de construir programas de simulação com filas, vamos sugerir um exercício que fará a simulação do que pode ocorrer nos sistemas operacionais multitarefas, para atender os vários processos (jobs) quando se dispõe apenas de um único processador (CPU). Um processo só consegue avançar com o seu processamento após alocar, ou seja, ter o processador à sua disposição para a realização das operações necessárias. Podemos visualizar o surgimento de um problema pela existência de um único processador: como o sistema operacional fará para que o uso do processador não seja monopolizado por um determinado processo? Lembre-se de que, como o ambiente é de multitarefas, poderá existir vários processos que necessitam alocar o processador.

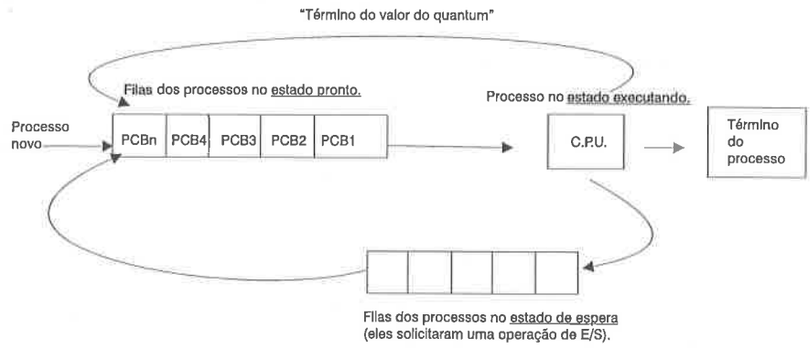
    Neste cenário descrito, o sistema operacional pode usar um algoritmo de escalonamento para controlar o uso do processador, denominado 'Round Robin' (algoritmo de revezamento). Com o uso desse algoritmo, o sistema operacional pode criar a 'ilusão' de atender a vários processos (tarefas requisitadas pelos usuários) de forma simultânea. O algoritmo RR usa uma característica denominada fatia de tempo (time-slice). Essa característica permite que o sistema operacional atenda cada um dos processos em um tempo determinado pelo valor de um parâmetro conhecido por 'quantum' (na ordem de milissegundos).

    Esse algoritmo é do tipo preemptivo, ou seja, o sistema operacional pode retomar o controle do processador quando ocorrer uma das três situações: término do processo, término do 'quantum' ou solicitação de uma operação de entrada/saída (leitura ou escrita de um registro em um arquivo ou até mesmo a digitação de um dado via teclado) por parte do processo que está utilizando o processador.

    O tamanho do 'quantum' determina o tempo máximo que um determinado processo pode utilizar o processador. Se no tempo do 'quantum' o processo não finalizar completamente ou não solicitar uma operação de E/S (o tempo de operação de leitura/gravação de um registro é muito demorado, quando comparado à velocidade de processamento do processador), o sistema operacional fará a preempção desse processo, retirando-o do processador e posicionando-o novamente na fila dos processos no estado pronto. Fazendo isso, o sistema operacional consegue realizar um atendimento mais eqüitativo dos vários processos que precisam usar o processador para evoluir a tarefa.

Ainda mais, no caso da operação de E/S, evita que o processador fique ocioso até que a operação de E/S seja, de alguma forma, finalizada. A seguir, um esquema ilustrativo do funcionamento do algoritmo RR.

Esquema ilustrativo do algoritmo de revezamento (*Round Robin*).



Em seguida veremos um diagrama de ação de estados que ilustra os principais estados que um processo pode assumir desde o momento em que ele é criado para o sistema. No momento em que um processo é criado, seu estado é 'pronto'. Ele é posicionado na fila de processos no estado pronto. Quando for permitido que o processo faça a alocação do processador, seu estado é alterado para 'executando'. Ao término do período estipulado pelo 'quantum', o processo é retirado do processador e retorna para a fila de processos no estado pronto, até chegar novamente a sua vez de alocar o processador. Se durante o seu estado 'executando', o processo realizar uma operação de E/S, o sistema operacional retira-o do processador, altera seu estado para 'espera' e coloca-o na fila de processos que estão aguardando o término da operação de E/S. O processo fica na fila de processos no estado 'espera', até que sua operação de E/S seja finalizada. Quando um processo finaliza totalmente sua tarefa, ele assume o estado de 'finalizado' e sai do sistema.

Diagrama de ação dos estados que um processo pode assumir no algoritmo RR.

Um processo pode ser materializado pelos sistemas operacionais através de uma estrutura denominada bloco de controle do processo (PCB — Process Control Block). Na prática, um PCB tem várias informações. Para fins didáticos, neste exercício você vai assumir que temos apenas quatro informações: número identificador do processo (gerado automaticamente pelo sistema operacional no momento da criação de um processo), tempo total que o processo precisa utilizar o processador para finalizar a sua tarefa, horário de entrada na fila dos processos do estado pronto e horário de término. Um vetor pode ser uma boa estrutura para representar um PCB com suas principais informações.

O programa a ser desenvolvido deve simular o cenário descrito acima, considerando-se uma fila para os processos no estado pronto e uma fila para os processos no estado de espera. Os seguintes eventos devem ser programados de forma aleatória:

- Criação de um processo;
- Solicitação de uma operação de E/S por parte dos processos, isto é, um processo pode ou não solicitar uma E/S enquanto estiver usando o processador;
- Momento de término de uma operação de E/S solicitada por um processo;
- Tempo total de uso do processador que o processo necessita para finalizar sua tarefa.

O valor do 'quantum' será fixo e previamente informado. À medida que os processos vão sendo finalizados, o programa deverá emitir um resumo do tempo que esses processos gastaram para serem totalmente atendidos (tempo atendimento = horário de término - horário de entrada), o tempo total que o processo permaneceu aguardando a vez para utilizar o processador (tempo total de espera = tempo na fila de estado pronto + tempo na fila de estado de espera) e o tempo acumulado de utilização do processador por todos os processos. Como desafio, procure realizar, de alguma forma, uma representação gráfica do funcionamento do algoritmo RR no atendimento dos processos.