

Onde estamos ...

1 Filas

2 Aplicações – Estudo de Casos

Capítulo 04 – Filas

Pontos fundamentais a serem cobertos:

- 1 Contexto e motivação
- 2 Definição
- 3 Implementações
- 4 Exercícios



- Assim como a estrutura de dados Pilha, Fila é outra estrutura de dados bastante utilizada em computação.
- Um exemplo é a implementação de uma fila de impressão.
- Se uma impressora é compartilhada por várias máquinas, normalmente adota-se uma estratégia para determinar a ordem de impressão dos documentos.
- A maneira mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

Aplicações

- 1 Escalonamento de processos na CPU (processos com a mesma prioridade) os quais são executados em ordem de chegada
- 2 Simulação de filas no mundo real tais como: filas em banco, compra de tickets, etc
- 3 Multi-programação (*time-sharing*)
- 4 Transferência assíncrona de dados (IO de arquivos, pipe, sockets)
- 5 Fila de espera em um *call-center*
- 6 Encontrar número de atendentes em supermercados e caixas bancários dado uma demanda de pessoas na sala de espera

Aplicações Indiretas

- ① Estrutura de dados auxiliares em algoritmos
- ② Componente de outras estruturas

Definição

Um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada de **início** da fila) e no qual podem-se inserir itens na outra extremidade (chamada **final** da fila).

Representação

- Os nós de uma fila são armazenados em endereços contínuos.
- A Figura 1 ilustra uma fila com três elementos.

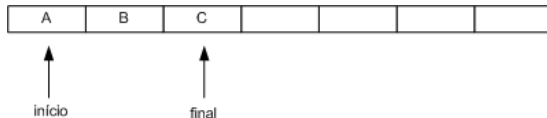


Figura 1: Exemplo de representação de fila.

- Após a retirada de um elemento (*primeiro*) temos:



Figura 2: Representação de uma fila após a remoção do elemento “A”.

Quais as estruturas usadas por filas?

- Baseada num vetor simples – limitada
- Baseada num vetor circular simples – limitada – igualmente poderosa
- Baseada num vetor circular dinâmico – **ilimitada**
- Baseada numa lista encadeada (depois de listas)

Representação

- Após a inclusão de dois elementos temos:

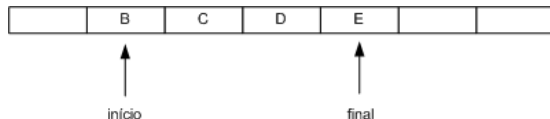


Figura 3: Representação de uma fila após a inclusão de dois elementos “D” e “E”.

- Como podemos observar, a operação de inclusão e retirada de um item da fila incorre na mudança do endereço do ponteiro que informa onde é o início e o término da fila.

Representação

- Em uma fila, o **primeiro** elemento inserido é o primeiro a ser removido.
- Por essa razão, uma fila é chamada **fifo** (*first-in first-out*) – primeiro que entra é o primeiro a sair – ao contrário de uma pilha que é **lifo** (*last-in, first-out*)
- Para exemplificar a implementação em C, vamos considerar que o conteúdo armazenado na fila é do tipo inteiro.
- Nos exemplos do prof é um outro tipo de dado
- A estrutura de fila possui a seguinte representação:

```
1 struct fila{
2     int elemento[N];
3     int ini;
4     int n; // quantos tem na fila
5 };
6
7 typedef struct fila Fila;
```

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si.
- Os membros são as variáveis *ini* e *fm*, que serve para armazenar respectivamente, o início e o fim da fila e o vetor *elemento* de inteiros que armazena os itens da fila.

Operações Primitivas

- As operações básicas que devem ser implementadas em uma estrutura do tipo Fila são:

Operação	Descrição
criar()	aloca dinamicamente a estrutura da fila.
insere(f,e)	adiciona um novo elemento (e), no final da fila f .
retira(f)	remove o elemento do início da fila f .

Tabela 1: Operações básicas da estrutura de dados fila.

Operações Auxiliares

- Além das operações básicas, temos as operações “*auxiliares*”. São elas:

Operação	Descrição
vazia(f)	informa se a fila está ou não vazia.
libera(f)	destrói a estrutura, e assim libera toda a memória alocada.

Tabela 2: Operações auxiliares da estrutura de dados fila.

Interface do Tipo Fila

```
1 typedef struct fila Fila;
2 /* Aloca dinamicamente a estrutura Fila, inicializando seus
3  * campos e retorna seu ponteiro. A fila depois de criada
4  * estarah vazia.*/
5 Fila* criar(void);
6
7 /* Insere o elemento e no final da fila f, desde que,
8  * a fila nao esteja cheia.*/
9 void insere(Fila* f, int e);
10
11 /* Retira o elemento do inicio da fila, e fornece o
12  * valor do elemento retirado como retorno, desde que a fila
13  * nao esteja vazia*/
14 int retira(Fila* f);
15
16 /*Verifica se a fila f estah vazia*/
17 int vazia(Fila* f);
18
19 /*Libera a memoria alocada pela fila f*/
20 void libera(Fila* f);
```

- Como em pilhas, e se estas estruturas puderem ser reutilizadas em diversos tipos de problemas?
- Ou seja, o reuso das mesmas estruturas de dados em problemas diferentes!
- O que é isto?
- Bem-vindos ao Tipos Abstractos de Dados (TAD ou TDA)
- Coloque as declarações de funções em um arquivo `.h` (Veja `stdio.h`)
- Coloque as funções em arquivos `.c`
- Deixe os arquivos `.o` para serem *linkeditados* no código relocável!
- Basicamente fizemos isto com o *Makefile*

Implementação de Fila com Vetor

- Assim como nos casos da pilha e lista, a implementação de fila será feita usando um vetor para armazenar os elementos.
- Isso implica, que devemos fixar o número máximo de elementos na fila.
- O processo de inserção e remoção em extremidades opostas fará a fila “*andar*” no vetor.
- Por exemplo, se inserirmos os elementos 8, 7, 4, 3 e depois retiramos dois elementos, a fila não estará mais nas posições iniciais do vetor.

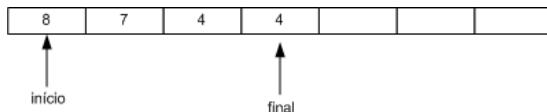


Figura 4: Fila após inserção de quatro elementos (um erro na figura 4/3)

Implementação de Fila com Vetor

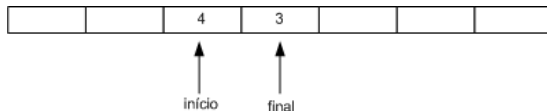


Figura 5: Fila após retirar dois elementos.

- Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada pelo vetor pode chegar a última posição.
- Uma solução seria ao remover um elemento da fila, deslocar a fila inteira no sentido do início do vetor.
- Entretanto, essa método é bastante ineficiente, pois cada retirada implica em deslocar cada elemento restante da fila. Se uma fila tiver 500 ou 1000 elementos, evidentemente esse seria um preço muito alto a pagar.

Implementação de Fila com Vetor

- Para reaproveitar as primeiras posições do vetor sem implementar uma “*re-arrumação*” dos elementos, podemos incrementar as posições do vetor de forma “*circular*”.
- Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam “*circularmente*”.
- Dessa forma, se temos 100 posições no vetor, os índices assumem os seguintes valores:

$0, 1, 2, 3, \dots, 98, 99, 0, 1, 2, 3, \dots, 98, 99, \dots$

Porquê um *vetor circular*?

Reflexões

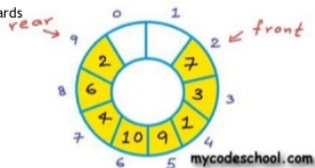
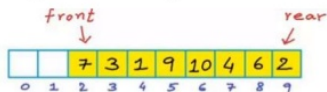
- Tamanho fixo – muito interessante para maioria dos problemas
- Velocidade
- Fácil de manipular
- *Circular* é a manipulação!
- Calcule $f(i + 1)$ dado que a cabeça ou fim de estivessem na posição i em um vetor de N posições (adote a convenção no sentido-horário)
- Cálculo com papel e caneta mesmo!

Queue: implementation Cyclic Array

no end on array
wrapping around
ring buffer / circular buffer
push/pop - $O(1)$

when item inserted to rear, tail's pointer moves upwards
when item deleted, head's pointer moves downwards

$current_position = i$
 $next_position = (i + 1) \% N$
 $prev_position = (N + i - 1) \% N$



Vetor Circular: Explicações

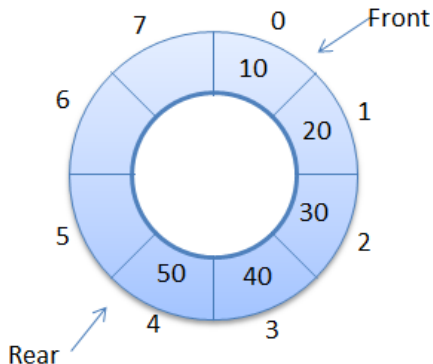
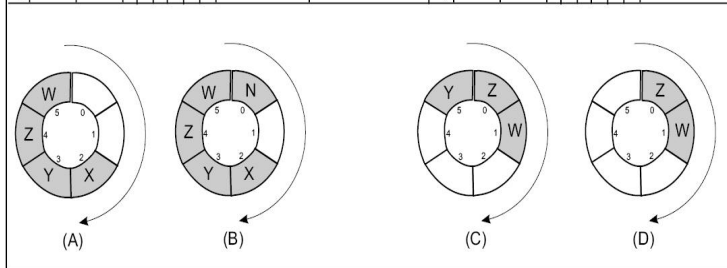


Figura 6: Notação aqui empregada

Vetor Circular: Explicações

	Frente	Cauda	Vetor						Operação
			0	1	2	3	4	5	
A)	2	5			X	Y	Z	W	Inserir N
B)	2	0	N		X	Y	Z	W	

	Frente	Cauda	Vetor						Operação
			0	1	2	3	4	5	
C)	5	1	Z	W				Y	Retira da Fila
D)	0	1	Z	W					



Função de Criação

- A função que cria uma fila, deve criar e retornar o ponteiro de uma fila vazia
- A função deve informar onde é o início da fila, ou seja, fazer $f \rightarrow ini = 0$, como podemos ver no código abaixo
- A complexidade de tempo para criar a fila é constante, ou seja, $O(1)$

```
1 /* Aloca dinamicamente a estrutura Fila, inicializando seus
2  * campos e retorna seu ponteiro. A fila depois de criada
3  * estarah vazia.
4  */
5 Fila* criar(void)
6 {
7     Fila* f = malloc(sizeof(Fila));
8     f->n = 0; // quantidade CORRENTE elementos
9     f->ini = 0; // inicio
10    return f;
11 }
```

Função de Inserção

- Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por **n**.
- Devemos assegurar que há espaço para inserção do novo elemento no vetor, haja vista se tratar de um vetor com capacidade limitada.
- A complexidade de tempo para inserir um elemento na fila é constante, ou seja, $O(1)$.

```
1 /* Insere o elemento e no final da fila f.*/  
2 void insere(Fila* f, int e)  
3 {  
4     int fim;  
5     if (f->n == N){  
6         printf("Fila cheia!\n"); }  
7     else{  
8         fim = (f->ini + f->n) % N;  
9         f->elementos[fim] = e;  
10        f->n++;  
11    }  
12 }
```


Função de Remoção

- A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno.
- Para remover um elemento, devemos verificar se a fila está ou não vazia.
- A complexidade de tempo para remover um elemento da fila é constante, ou seja, $O(1)$.

```
1 int retira(Fila* f)
2 {
3     int e;
4     if ( vazia(f) )
5         printf("Fila vazia!\n");
6     else{
7         e = f->elementos[f->ini];
8         f->ini = (f->ini + 1) % N;
9         f->n--;
10    }
11    return e;
12 }
```

Exemplo de Uso da Fila

```
1 #define N 10
2 #include <stdio.h>
3 #include "fila.h" // TDA.h
4
5 int main(void)
6 {
7     Fila * f = criar();
8
9     int i;
10    for (i = 0; i < N; i++)
11        insere(f, i * 2);
12
13    printf("\nElementos removidos: ");
14
15    for (i = 0; i < N/2; i++)
16        printf("%d ", retira(f));
17
18 }
```

Onde estamos ...

1 Filas

2 Aplicações – Estudo de Casos

Fila de Prioridade

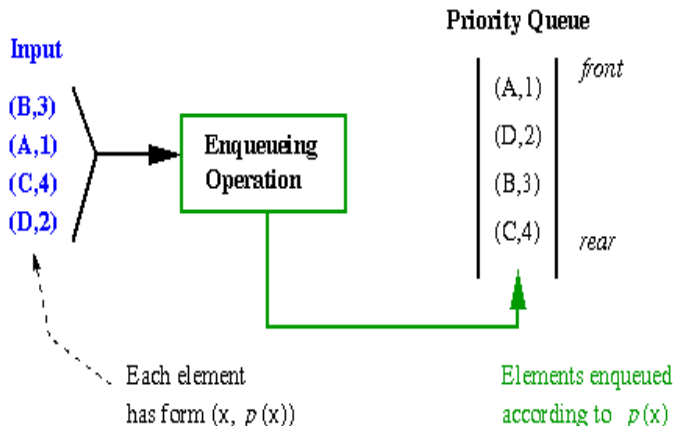


Figura 7: Várias aplicações para um único atendedor

Filas de Prioridade – Várias

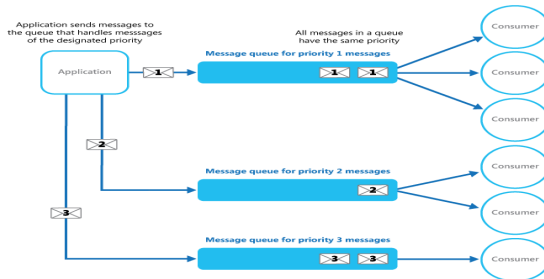


Figura 8: Filas com prioridades e múltiplos atendedores

Filas de Prioridade – Uso de CPU

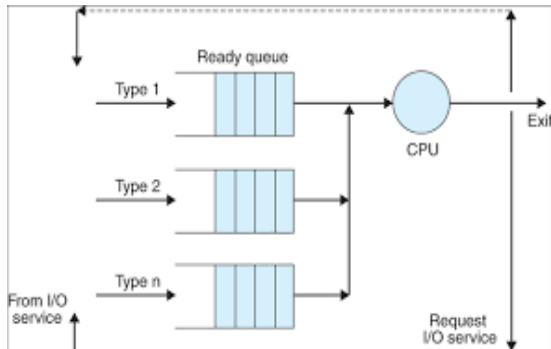


Figura 9: Filas com prioridades – clássico *round-robin*

Filas de Prioridade – Múltiplos Atendedores

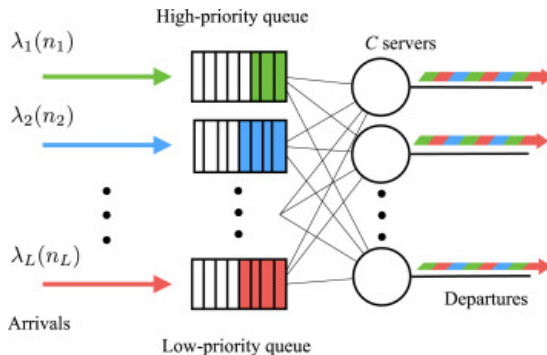


Figura 10: Filas com prioridades – clássico *round-robin*

Filas com Múltiplos Atendedores

BFS: Busca em Largura

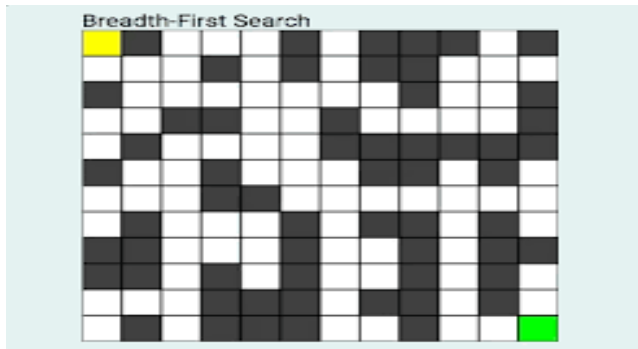


Figura 11: Casos clássicos – pesquisas em labirinto – grafos

Comparativo BFS X DFS *breadth first search (BFS)* *and depth first search (DFS)*

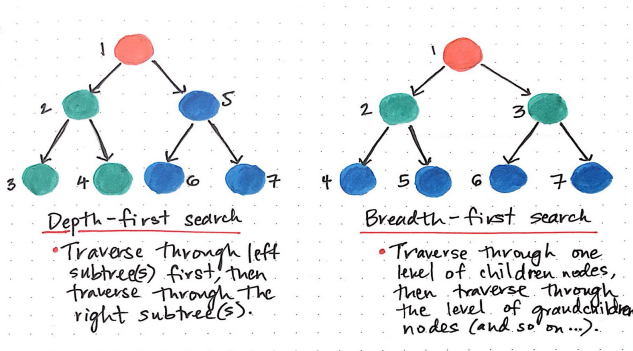


Figura 12: Casos clássicos – pesquisas em labirinto – grafos

BFS *breadth first search* (BFS) em Grafo

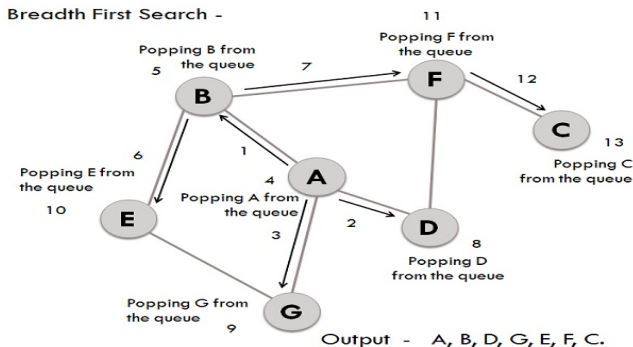


Figura 13: Caso clássico – grafos

A árvore de busca do BFS (*breadth first search*) em Grafos

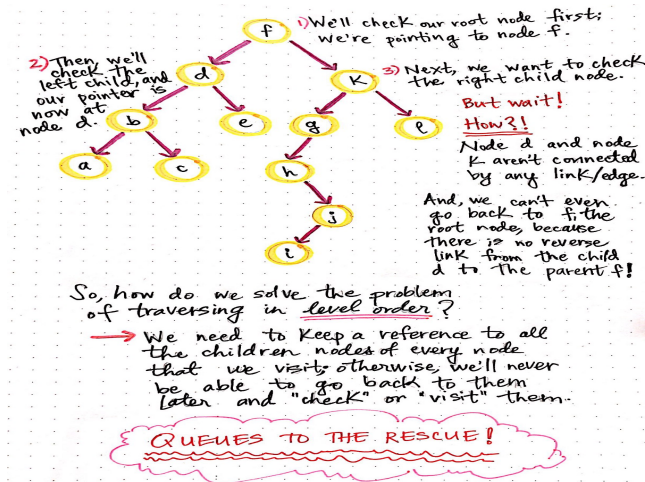


Figura 14: A árvore de busca do BFS

A árvore de busca do BFS (*breadth first search*) em Grafos

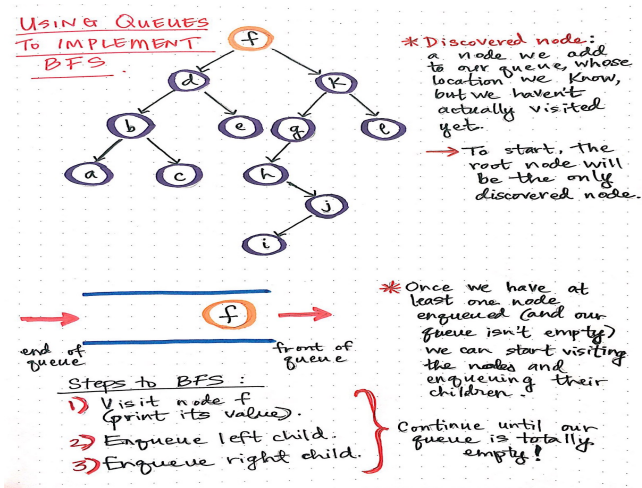
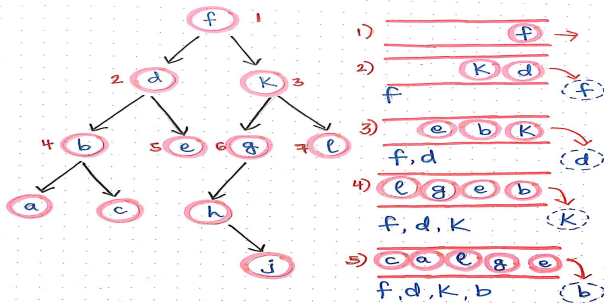


Figura 15: A árvore de busca do BFS

A árvore de busca do BFS (*breadth first search*) em Grafos



But what about space-time complexity?

- Visiting a node (reading its data and enqueueing its children) takes constant time. Since we are only visiting each node once, the time it will take us to use a BFS is $O(n)$, where n is the number of nodes.
- The space complexity depends on the size of the queue at its worst, which could be up to $O(n)$ also.

Figura 16: A árvore de busca do BFS

Pseudo Código do BFS (*breadth first search*)

Busca em Largura usando Estrutura de Fila (P)

pseudo-código by CCS

Pré-requisitos:

- Leitura de uma matriz de adjacência G
- Uma fila F e seus métodos – chegada (push) e partida (pop)
- Um vetor de status de n-vértices

varredura_do_grafo(G)

```
{
  vetor_status_nós[i] = aberto; // todos os i-nós marcados como não visitados
  no_inicial = lê_no_inicial();
  BFS( no_inicial );           // chama BFS
}
```

BFS(vert)

```
{
  chegada( vert, F);           // vertice = nó inicial da busca
  enquanto ( !vazia(F) )      // enquanto fila não estiver vazia
  {
    x = partida( F);           // obtém um vértice x a ser inspecionado
    se ( vetor_status_nós[x] == aberto )
    {
      imprime_ou_visita(x);
      teste_o_que_quiseres(x); // AQUI se testa x ...
      vetor_status_nós[x] = fechado; // nó visitado, muda status
    };
    para todos vértices de i = x até x >= 0 faça
    // aqui muda o sentido da busca ... esquerda ou direita
    {
      se ( (vetor_status_nós[i] == aberto ) AND
            (vertice_vizinho[i] == 1) )
      chegada(i,F);
    }; // fim do empilhamento dos nos adjacentes novos
  }; // fim do enquanto
fim_do_BFS;
};
```


Exercícios

- 1 Implemente um esquema de uma fila usando duas pilhas
- 2 Encontrar o maior sub-conjunto de uma janela w para um vetor circular N . Basicamente, preencher o vetor inteiro, posicionar início em fim de fila, tal que a diferença seja w . Rotacione com acionamentos de chegadas e partidas, até encontrar a maior soma neste vetor. Ou seja, $|\text{rear} - \text{front} + 1| = w$. Função:
 $\text{abs}(\text{rear} - \text{front} + 1) = w$
- 3 Construa uma função que copie uma fila, para uma nova fila invertida.
- 4 Faça uma simulação aleatória de chegadas e partidas em uma fila, tal que a cada passo exiba as mensagens de fila cheia, fila vazia, número de elementos corrente na fila. Faça isto para um número considerável de simulações. Este exercício é a base na simulação de sistemas de filas.
- 5 Implemente um sistema de fila com prioridades (será um dos projetos)

- ① Karumanchi, Narashimha (2017). *Data Structures and Algorithms Made Easy – Data Structures and Algorithms and Puzzles*. CareerMonk.com
- ② Tenenbaum, A. M., Langsam, Y., and Augestein, M. J. (1995). *Estruturas de Dados Usando C*. MAKRON Books, pp. 207-250.
- ③ Wirth, N. (1989). *Algoritmos e Estrutura de Dados*. LTC, pp. 151-165.