

SCE 202 – Algoritmos e Estruturas de Dados I

Filas





Conceito

- Coleção ordenada de itens (lista ordenada) em que a inserção de um novo item se dá em um dos lados – no fim – e a remoção no outro lado – no início.
 - Listas FIFO/LILO (**F**irst **I**n **F**irst **O**ut/ **L**ast **I**n **L**ast **O**ut).
- Modelos intuitivos de filas são as linhas para comprar bilhetes de cinema e de caixa de supermercado.
- A fila, como a pilha, é conceitualmente uma estrutura dinâmica que está continuamente mudando pois itens são adicionados/retirados.

TAD – Fila – Operações

```
void definir (fila *q);
```

```
/*Cria uma fila vazia. Deve ser usado antes de  
qualquer outra operação*/
```

```
void tornar_vazia (fila *q);
```

```
/*Reinicializa uma fila existente, q, como uma  
fila vazia. Dependendo da implementação da  
estrutura de dados, deve remover todos os seus  
elementos.*/
```

```
boolean vazia (fila *q);
```

```
/*Retorna true se fila não contém elementos, false  
caso contrário*/
```

```
boolean inserir (fila *q, tipo_info item);
```

```
/*Adiciona um item no fim da fila q. Retorna true  
se operação realizada com sucesso, false caso  
contrário*/
```



TAD – Fila – Operações

```
boolean remover(fila *q, tipo_info *item);  
/*Remove um item do início da fila q. Retorna true  
se operação realizada com sucesso, false caso  
contrário*/
```

```
int tamanho (fila *q);  
/*Retorna o tamanho da fila*/
```

```
boolean começo_fila (fila q, tipo_info *item);  
/*Mostra o começo da fila sem remover o item.  
Retorna true se operação realizada com sucesso,  
false caso contrário*/
```



Implementações de Filas: Estática

Há um meio de se utilizar de um array na implementação de uma fila?

SIM, se nós dimensionarmos o array com um tamanho que dê para acomodar o tamanho máximo da fila, e além disso precisamos dos ponteiros FIM e COMEÇO.

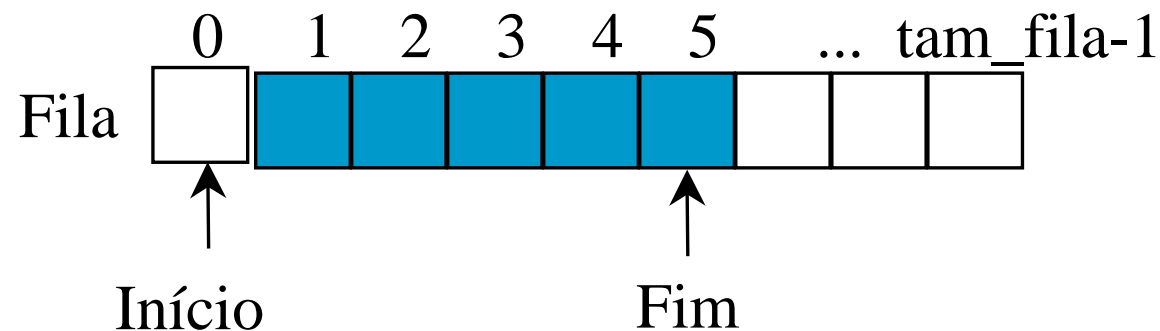
Implementações de Filas: Estática

```
#define tam_fila 100  
#define indice int
```

```
typedef struct{  
    tipo_info A[tam_fila];  
    indice inicio, fim;  
}fila;
```

inicio: aponta para a posição **anterior** ao 1º elemento.

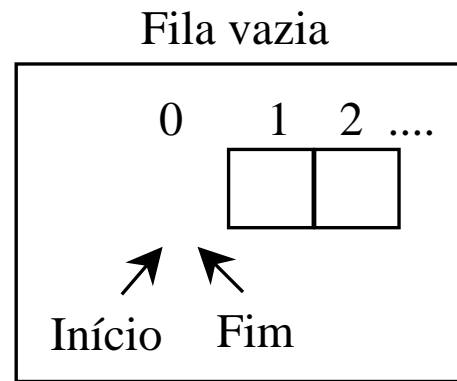
fim: aponta para a posição do **último** elemento.



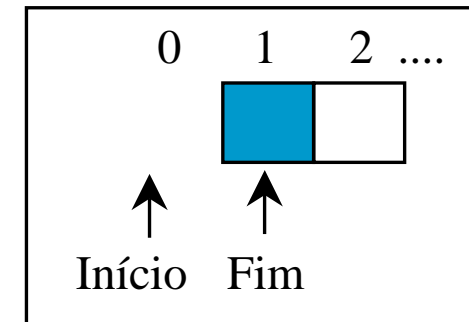
Implementações de Filas: Estática

O que é então uma fila vazia?

No começo: $\text{inicio} = \text{fim} = 0$

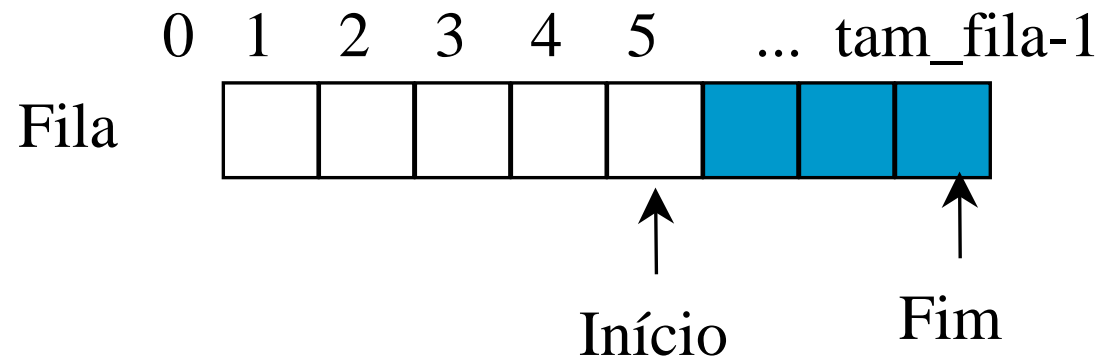


Fila com um elemento



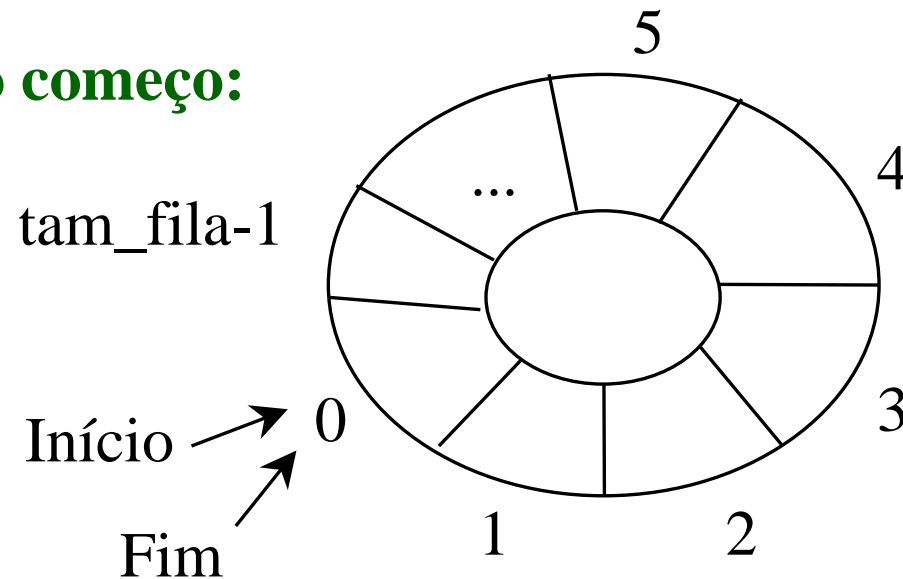
Num instante qualquer: $\text{Início} = \text{Fim}$

E o que é uma fila cheia? $\text{Fim} = \text{tam_fila?}$



Solução: Fila do Tipo Anel

No começo:



fila vazia =
(Início=Fim=0)

Inserir: incrementa Fim, se não estiver cheia

Eliminar: se não estiver vazia (Início=Fim), incrementa Início.

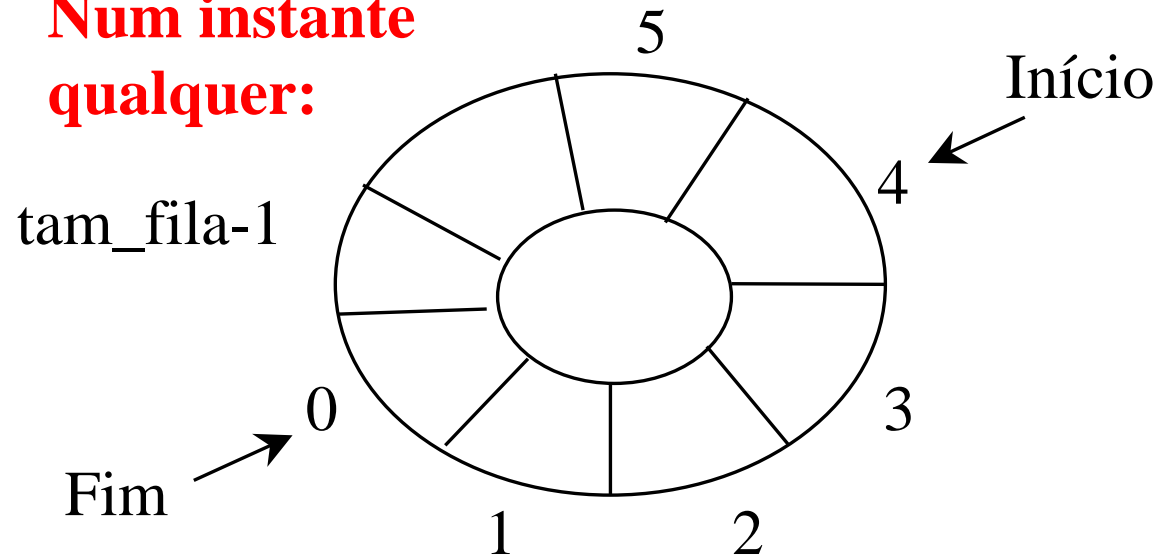
Incrementar, no Anel, implica ignorar limite do tam_fila:

Início = (Início + 1) % tam_fila

Fim = (Fim + 1) % tam_fila

Fila tipo Anel

Num instante qualquer:



Se permitirmos o uso da posição Início para inserção, as condições de fila cheia e vazia seriam idênticas.

Providência: uma posição é sacrificada; apenas $\text{tam_fila}-1$ posições são utilizadas pela fila.

Assim:

Condição de Fila Cheia = $(\text{Fim} + 1) \% \text{tam_fila} = \text{Início}$

Condição de Fila Vazia = $\text{Fim} = \text{Início}$



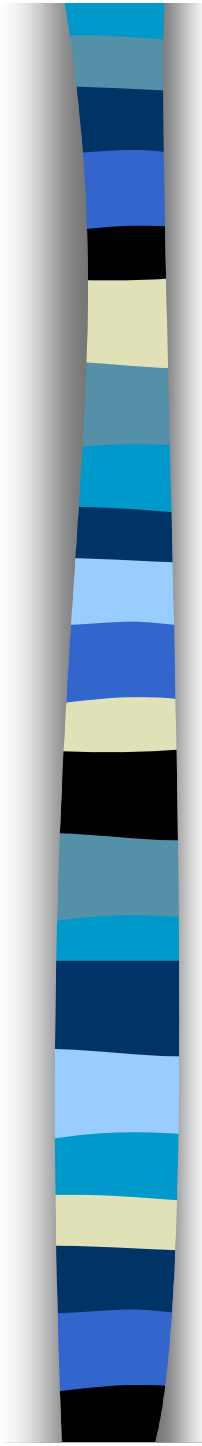
Implementações de Filas: Circular Estática

```
#define tam_fila 100  /*nº máx. itens na fila*/
#define indice int

/*permite um espaço em branco para diferenciar lista
   cheia de vazia*/

typedef struct{
    tipo_info A[tam_fila];
    indice inicio, fim;
}fila;

fila q; /*tipo de declaração*/
```



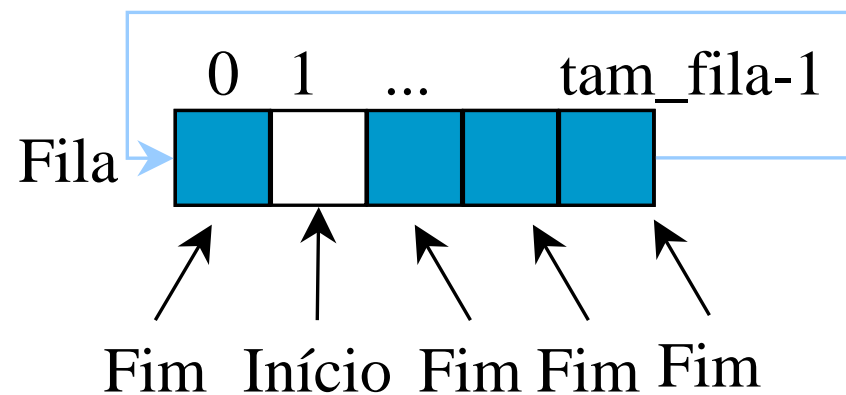
```
void definir (fila *q){
    /*Cria uma fila vazia. Deve ser usado antes de qualquer outra
    operação*/
    q->fim = 0;
    q->inicio = 0;
    /*ponteiro de início atrasado; aponta para uma posição
    anterior ao início*/
}

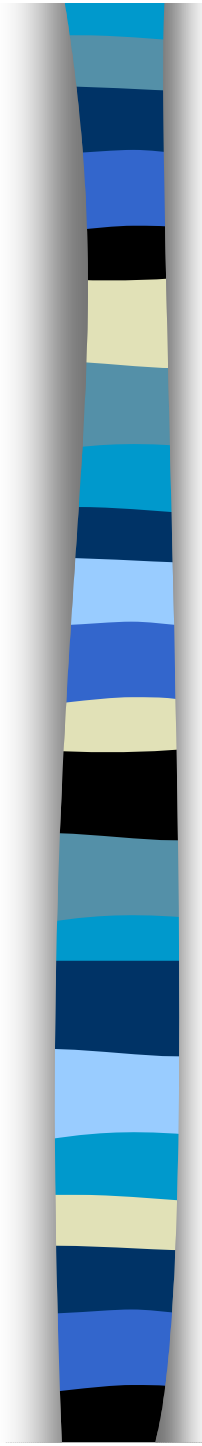
boolean vazia (fila *q){
    /*Retorna true se fila não contém elementos, false caso
    contrário*/
    return (q->inicio == q->fim);
}

boolean cheia (fila *q){
    /*Retorna true se fila cheia, false caso contrário*/
    return (q->inicio == ((q->fim + 1) % tam_fila));
    /*os dois ponteiros diferem de uma posição*/
}
```

```
boolean inserir (fila *q, tipo_info item){
    /*Adiciona um item no fim da fila q. Retorna true se
    operação realizada com sucesso, false caso contrário*/
    /*uma posição da fila nunca será preenchida*/
    if (cheia(*q))
        return FALSE;

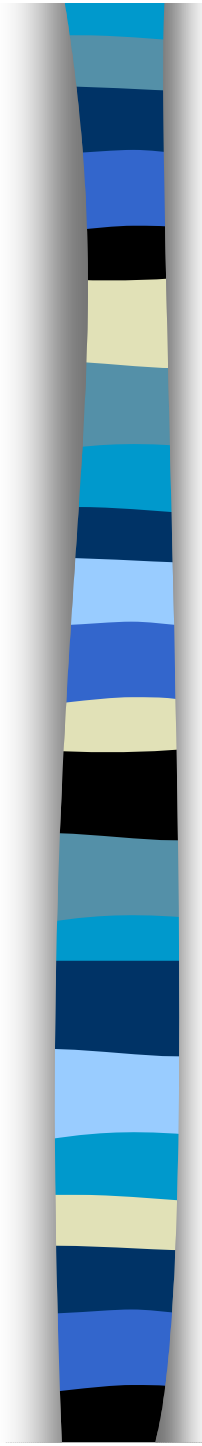
    q->fim = (q->fim + 1) % tam_fila ;
    q->A[q->fim] = item;
    return TRUE;
}
```





```
boolean remover(fila *q, tipo_info *item){
    /*Remove um item do início da fila q. Retorna true se
    operação realizada com sucesso, false caso contrário*/
    if (vazia(*q))
        return FALSE;

    q->inicio = (q->inicio+1) % tam_fila;
    item = q->A[q->inicio]; /*opcional*/
    return TRUE;
}
```



```
int tamanho (fila *q){
    /*retorna o tamanho da fila*/
    if (q->inicio <= q->fim)
        return (q->fim - q->inicio);

    return (tam_fila - (q->inicio - q->fim));
}

boolean começo_fila (fila *q, tipo_info *item){
    /*Mostra o começo da fila sem remover o item. Retorna
    true se operação realizada com sucesso, false caso
    contrário*/
    if (vazia(*q))
        return FALSE;

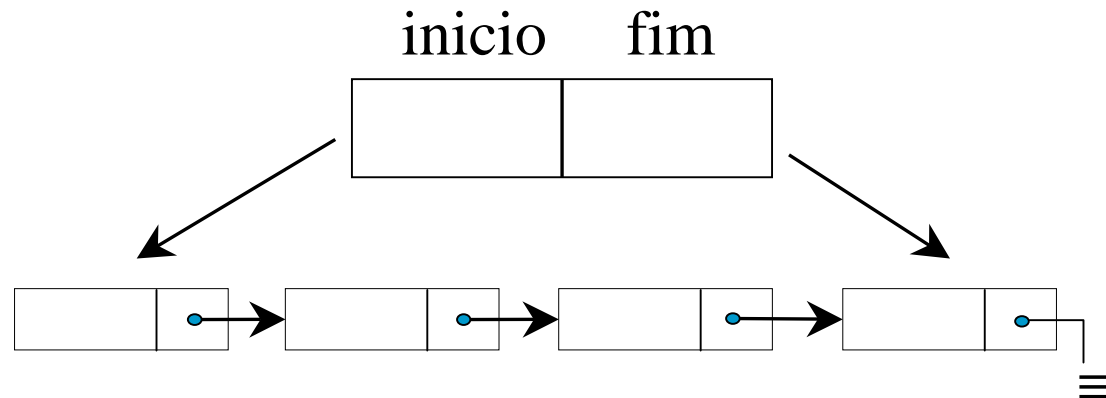
    item = q->A[(q->inicio+1) % tam_fila];
    return TRUE;
}
```

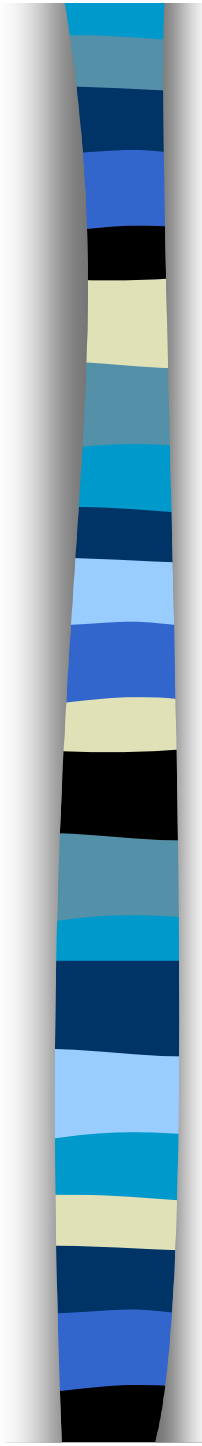
Implementações de Filas: Dinâmica

```
#define tipo_info int
```

```
typedef struct elem{  
    tipo_info info;  
    struct elem *lig;  
}tipo_elem;
```

```
typedef struct{  
    tipo_elem *inicio;  
    tipo_elem *fim;  
}fila;  
fila q;
```





```
void definir(fila *q){
    /*Cria uma fila vazia. Deve ser usado antes de qualquer
    outra operação*/
    q->inicio = NULL;
    q->fim = NULL;
}

boolean vazia (fila *q){
    /*Retorna true se fila não contém elementos, false caso
    contrário*/
    return (q->inicio == NULL);
}

void tornar_vazia (fila *q){
    /*Reinicializa uma fila existente q como uma fila vazia
    removendo todos os seus elementos.*/
    tipo_elem *ndel, *nextno;

    if(!vazia(q)){
        nextno = q->inicio;
        while (nextno != NULL){
            ndel = nextno;
            nextno = nextno->lig;
            free(ndel);
        }
    }
    definir(q);
}
```



```

boolean inserir (fila *q, tipo_info info){
    /*Adiciona um item no fim da fila q. Retorna true se
    operação realizada com sucesso, false caso contrário*/
    tipo_elem *p;

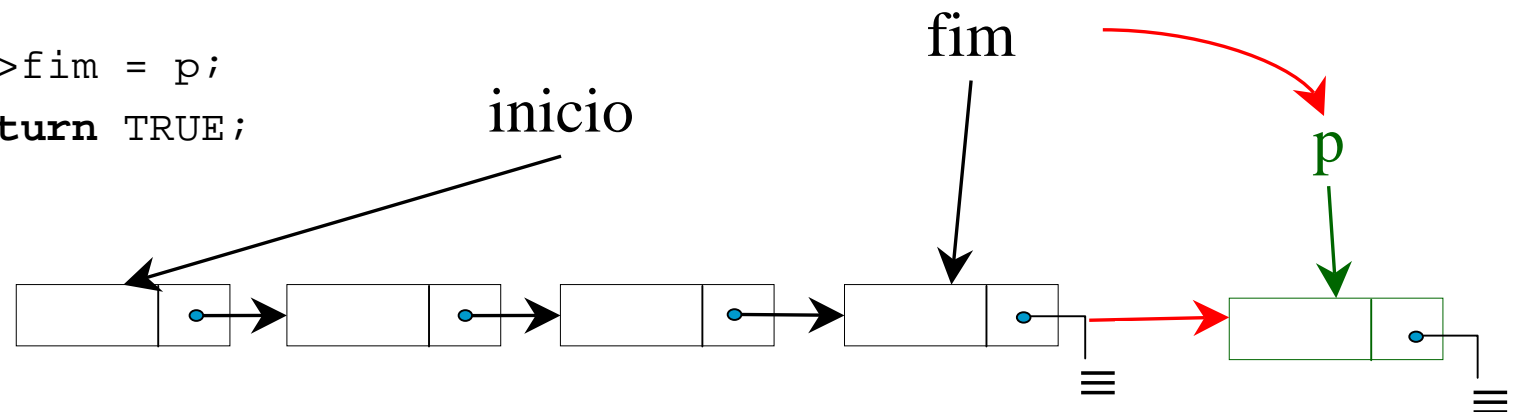
    p = malloc(sizeof(tipo_elem));
    if (p == NULL)
        return FALSE;

    p->info = info;
    p->lig = NULL;

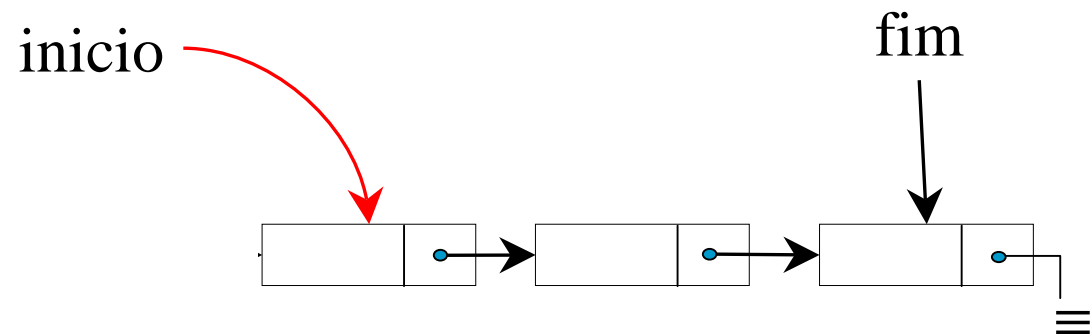
    if (vazia(q))
        q->inicio = p;
    else
        q->fim->lig = p;

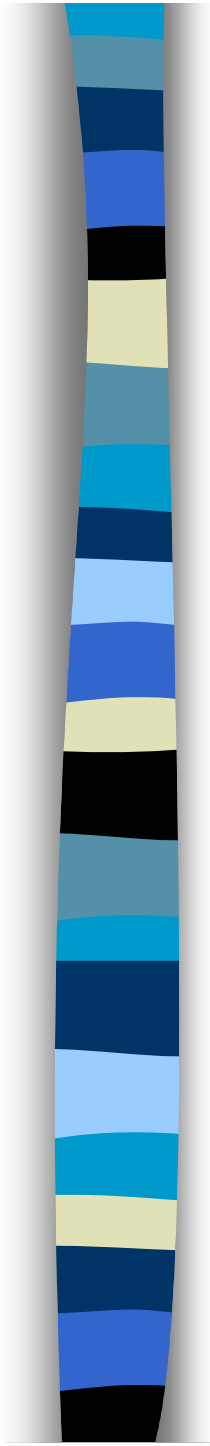
    q->fim = p;
    return TRUE;
}

```



```
boolean remover(fila *q, tipo_info *info){  
    /*Remove um item do início da fila q. Retorna true se  
    operação realizada com sucesso, false caso contrário*/  
    tipo_elem *p;  
  
    if (vazia(q))  
        return FALSE;  
  
    p = q->inicio;  
    *info = p->info;  
    q->inicio = p->lig;  
  
    if (q->inicio == NULL)  
        q->fim = NULL;  
  
    free(p);  
    return TRUE;  
}
```





```
int tamanho (fila *q){
    /*Retorna o tamanho da fila*/
    tipo_elem *p;
    int cont = 0;

    p = q->inicio;
    while (p != NULL){
        cont ++;
        p = p->lig;
    }

    return cont;
}
```

```
boolean começo_fila (fila *q, tipo_info *item){
    /*Mostra o começo da fila sem remover o item. Retorna true
    se operação realizada com sucesso, false caso contrário*/
    if (vazia(q))
        return FALSE;

    *item = q->inicio->info;
    return TRUE;
}
```



Análise dos 2 tipos de Representação

- Vantagens da Fila Estática (Anel):
 - não envolve custos da alocação dinâmica
- Desvantagens da Fila Estática:
 - previsão de tamanho máximo
- Vantagens da Fila Dinâmica:
 - ocupa espaço estritamente necessário
- Desvantagens da Fila Dinâmica:
 - custos usuais da alocação dinâmica (tempo de alocação, campos de ligação)



Quando usar

- Representação Estática (Anel):
 - quando fila tiver tamanho pequeno ou seu comportamento for previsível
- Representação Dinâmica:
 - nos demais casos



Exercícios

1. Implemente um procedimento reverso que reposiciona os elementos na fila de forma que o início se torne fim e vice-versa. Use uma pilha.
 - $I \quad F \rightarrow I \quad F$
 - $1 \rightarrow 2 \rightarrow 3 \quad 3 \rightarrow 2 \rightarrow 1$
2. Obtenha uma representação mapeando uma pilha P e uma fila F em um único array V[n]. Escreva algoritmos para inserir e eliminar elementos destes 2 objetos de dados. O que você pode dizer sobre a conveniência de sua representação?



Filas de Prioridade

- Filas em que a prioridade de remoção não é cronológica
 - Maior prioridade não é do elemento que ingressou primeiro
- Exemplos de Aplicações
 - Vôos lotados (*standby flyers*)
 - Listas de espera em geral (p. ex. transplantes)
 - Fila de processos para o Sistema Operacional
 - Ordenação



TAD Fila de Prioridade

- Armazena **Itens**
- **Item**: par (chave, informação)
- Operações principais:
 - **remove**(F): remove e retorna o item com maior prioridade (menor ou maior chave) da fila F
 - **insert**(F, **x**): insere um item **x** = (k,e) com chave k
- Operações auxiliares:
 - **get**(F): retorna o item com menor (maior) chave da fila F, sem removê-lo
 - **size**(F), **isEmpty**(F)



TAD Fila de Prioridade

◆ Diferentes Realizações

■ Estáticas

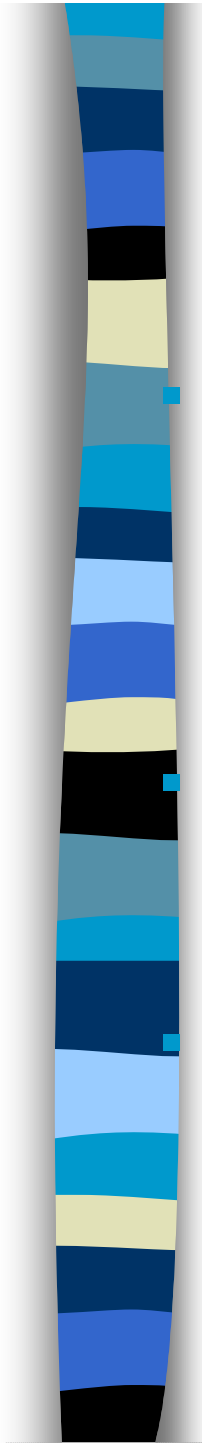
- ◆ Lista estática (array) ordenada
- ◆ Lista estática (array) não ordenada

■ Dinâmicas

- ◆ Lista dinâmica ordenada
- ◆ Lista dinâmica não ordenada

◆ Cada realização possui vantagens e desvantagens

Chaves e Relações de Ordem Total



- **Chaves** em uma lista de prioridade podem ser objetos arbitrários para os quais uma ordem é definida

- Podem ser bem mais complexos que um simples valor numérico

- Dois itens distintos em uma fila de prioridade podem ter chaves iguais

- Definição matemática de uma **relação de ordem total** (\leq):

- Propriedade Reflexiva:

$$a \leq a$$

- Propriedade Anti-simétrica:

$$a \leq b \text{ e } b \leq a \Rightarrow a = b$$

- Propriedade Transitiva:

$$a \leq b \text{ e } b \leq c \Rightarrow a \leq c$$

TAD Comparador

Um comparador encapsula a ação de comparar dois objetos de acordo com uma dada relação de ordem total

Uma fila de prioridade *genérica* deve *receber* um comparador para as chaves que irá utilizar:

- Ponteiro para função (C, C++)
- Objeto comparador (Java, C++)

O comparador é acionado pela fila de prioridade e recebe desta as chaves a serem comparadas

■ Operação primária:

- `compare(a, b)`: retorna um número i tal que $i < 0$ se $a < b$, $i = 0$ se $a = b$, e $i > 0$ se $a > b$

■ Operações adicionais:

- `comparavel(a, C)`: retorna *true* se o tipo de **a** é compatível com aquele do comparador C, *false* caso contrário
- ...

Exemplo de Comparador

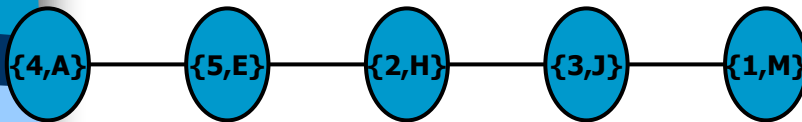
- Comparação Lexicográfica de pontos 2D em C:

```
typedef struct
    float x, y;
} Point2D;

float compare(Point2D *a, Point2D *b){
    if (a->x != b->x)
        return (a->x - b->x);
    else
        return (a->y - b->y);
}
```

Realização Baseada em Listas

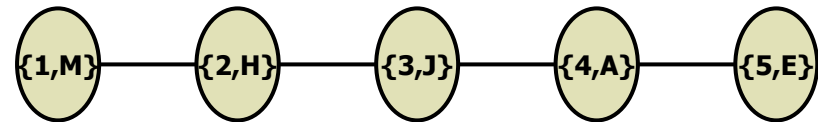
■ Estratégia 1: Utilizando uma **lista não ordenada**:



■ Desempenho:

- **insert** executa em tempo $O(1)$ dado que podemos inserir o item no início ou fim da lista.
- **remove** e **get** executam em tempo $O(n)$ dado que devemos percorrer a lista para encontrar a menor / maior chave.

■ Estratégia 2: Utilizando uma **lista ordenada**:



■ Desempenho:

- **insert** executa em tempo linear $O(n)$ dado que devemos encontrar o lugar correto onde inserir o item.
- **remove** e **get** executam em tempo $O(1)$ dado que a menor / maior chave está no início (ou fim) da lista.

PS 1. Assume-se que as chaves podem ser comparadas em tempo $O(1)$.

PS 2. Análise vale tanto para listas estáticas como para listas dinâmicas.

Aplicação: Ordenação via Filas de Prioridade

Podemos utilizar uma fila de prioridade para ordenar uma coleção de elementos comparáveis:

1. Insira os elementos na fila um a um via uma série de operações *insert* (Fase 1)
2. Retorne os elementos via uma série de operações *remove* (Fase 2)

O tempo de execução deste método depende da implementação da fila de prioridade

Algoritmo *PQ-Sort(L)*

Entrada: lista *L* de itens

Saída: lista *L* ordenada

F ← nova fila de prioridade

enquanto não *lista_vazia(L)*

x ← *Remover_frente(L)*

insert(F, x)

enquanto não *isEmpty(F)*

x ← *remove(F)*

Inserir_final(L, x)

Ordem crescente para fila ascendente (chave mínima) e
decrecente para fila descendente (chave máxima)



Aplicação: Ordenação via Filas de Prioridade

Selection-Sort é um algoritmo de ordenação que pode ser visto como uma variação de PQ-Sort:

- Fila de prioridade implementada via *lista não ordenada*
 - Estratégia 1 discutida anteriormente.
 - Não confunda a lista *L* de PQ-Sort com a lista da fila de prioridade !!

Tempo de execução:

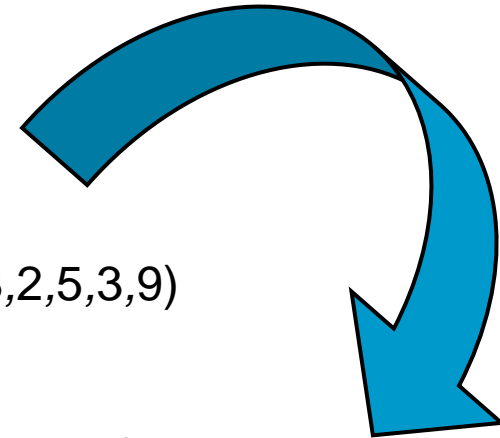
1. Fase 1: Inserir os elementos em uma fila de prioridade com n operações **insert** consome tempo $O(n)$
2. Fase 2: Remover os elementos a partir da fila de prioridade com n operações **remove** consome tempo:

$$O(n + n-1 + \dots + 2 + 1) = O(n(n+1)/2) = O(n^2)$$

Logo, a complexidade computacional do algoritmo em termos de tempo de execução é $O(n^2)$

Exemplo (Selection-Sort)

	<i>Lista L</i>	<i>Fila de Prioridade F</i>
Entrada:	(7,4,8,2,5,3,9)	()
Fase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..
.	.	.
(g)	()	(7,4,8,2,5,3,9)
Fase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()



1a Fase pode parecer inútil, mas serve para transformar a ED lista em ED fila de prioridade.