Filas

- Nesta aula veremos o ADT fila
- Em um computador existem muitas filas esperando pela impressora, acesso ao disco ou, num sistema *time-sharing*, pelo uso da CPU

Algoritmos e Estruturas de Dados I

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP

augusto@usp.br
http://dfm.ffclrp.usp.br/~augusto

Organização

- Definição do ADT Fila
- Especificação
  - Operações sobre o ADT Fila, utilizando pré- e pós-condições
- Implementação
  - Estática (contígua)
  - Dinâmica (encadeada)

Definição

- Uma **fila** (*queue*) é uma lista linear na qual remoções são realizadas em uma extremidade (início ou *front* ou *head*) e todas adições na lista são feitas em outra extremidade (final ou *rear* ou *tail*)

Definição

- Por exemplo, pense numa fila de pessoas para pegar o metrô
- As pessoas vão chegando...

Definição

- Por exemplo, pense numa fila de pessoas para pegar o metrô
- As pessoas vão chegando...

Definição

- Por exemplo, pense numa fila de pessoas para pegar o metrô
- As pessoas vão chegando...

Definição

- Por exemplo, pense numa fila de pessoas para pegar o metrô
- As pessoas vão chegando...

Definição

- Por exemplo, pense numa fila de pessoas para pegar o metrô
- As pessoas vão chegando...

Definição

- Por exemplo, pense numa fila de pessoas para pegar o metrô
- As pessoas vão chegando uma após a outra
- A primeira a chegar é a primeira a entrar no metrô
- A última a chegar é a última a entrar no metrô

Operações Fundamentais

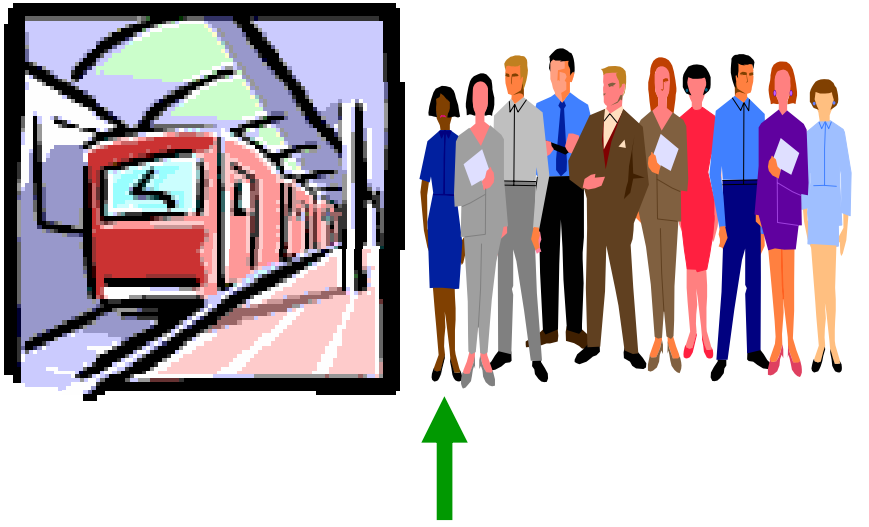
- Quando um item é adicionado numa fila, usa-se a operação **Append** (inserir no final)

Operações Fundamentais

- Quando um item é adicionado numa fila, usa-se a operação **Append** (inserir no final)

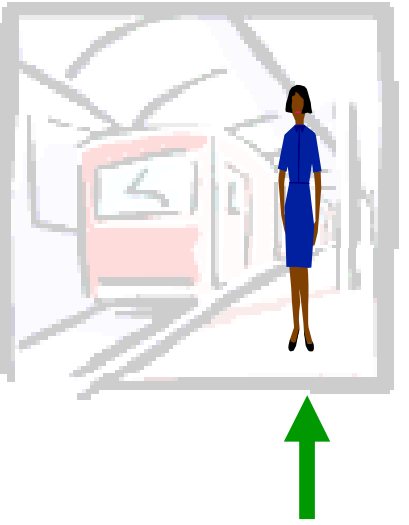
Operações Fundamentais

- Quando um item é adicionado numa fila, usa-se a operação **Append** (inserir no final)
- Quando um item é retirado de uma fila, usa-se a operação **Serve** (servir)



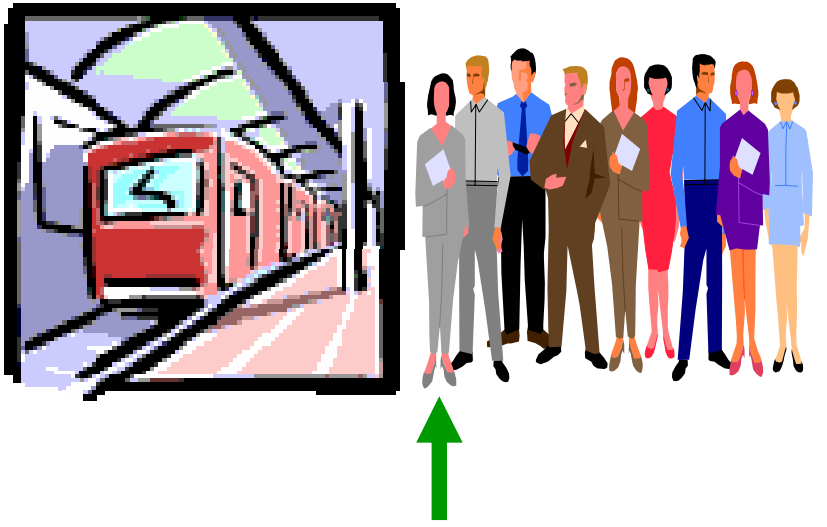
Operações Fundamentais

- Quando um item é adicionado numa fila, usa-se a operação **Append** (inserir no final)
- Quando um item é retirado de uma fila, usa-se a operação **Serve** (servir)



Operações Fundamentais

- Quando um item é adicionado numa fila, usa-se a operação **Append** (inserir no final)
- Quando um item é retirado de uma fila, usa-se a operação **Serve** (servir)

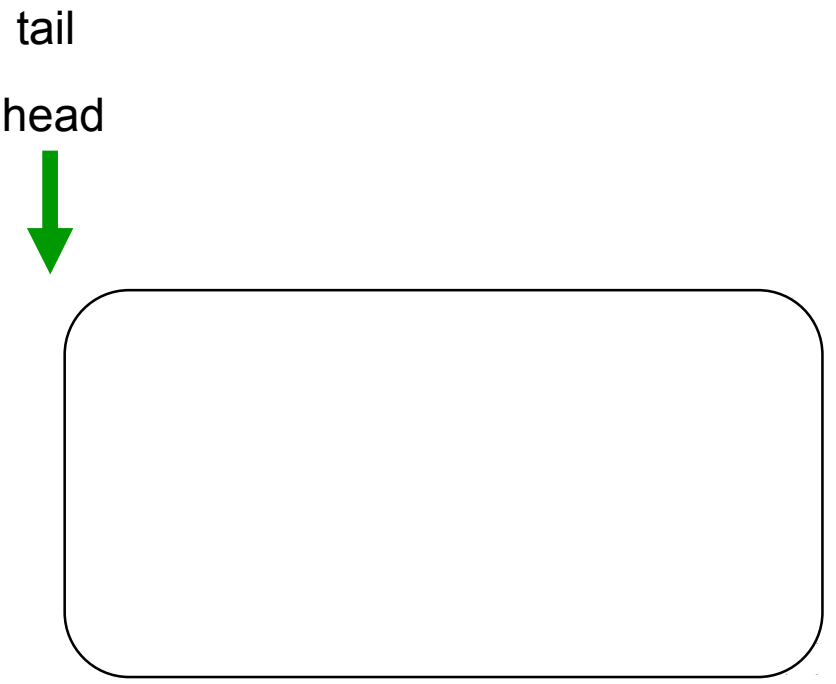


Operações Fundamentais

- O primeiro item inserido (**Append**) na fila é sempre o primeiro a ser retirado (**Serve**)
- Esta propriedade é denominada *First In, First Out* (primeiro a entrar, primeiro a sair) ou **FIFO**

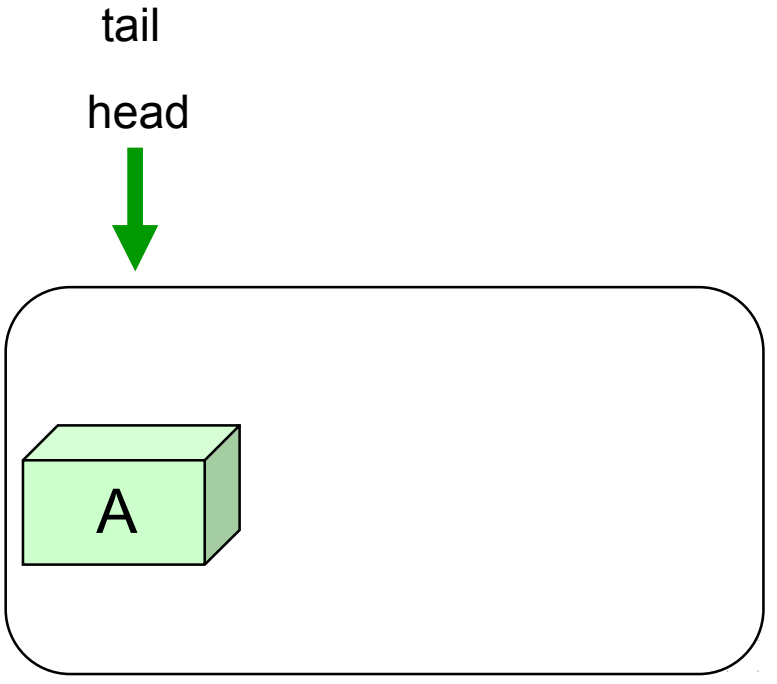
Exemplo

- fila vazia inicialmente



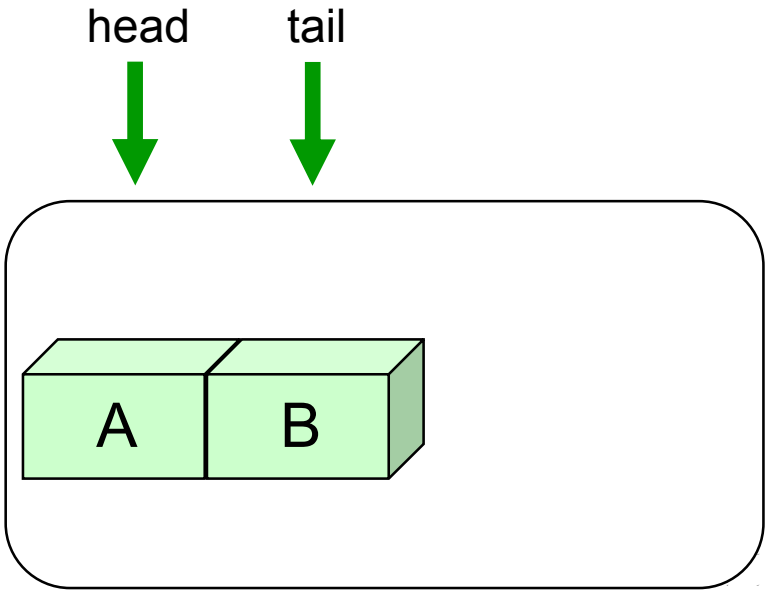
Exemplo

- fila vazia inicialmente
- inserir caixa A



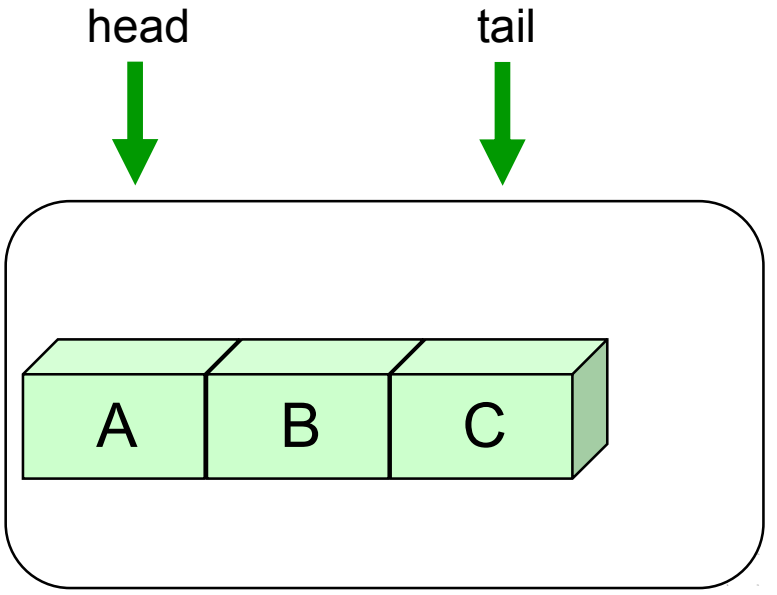
Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B



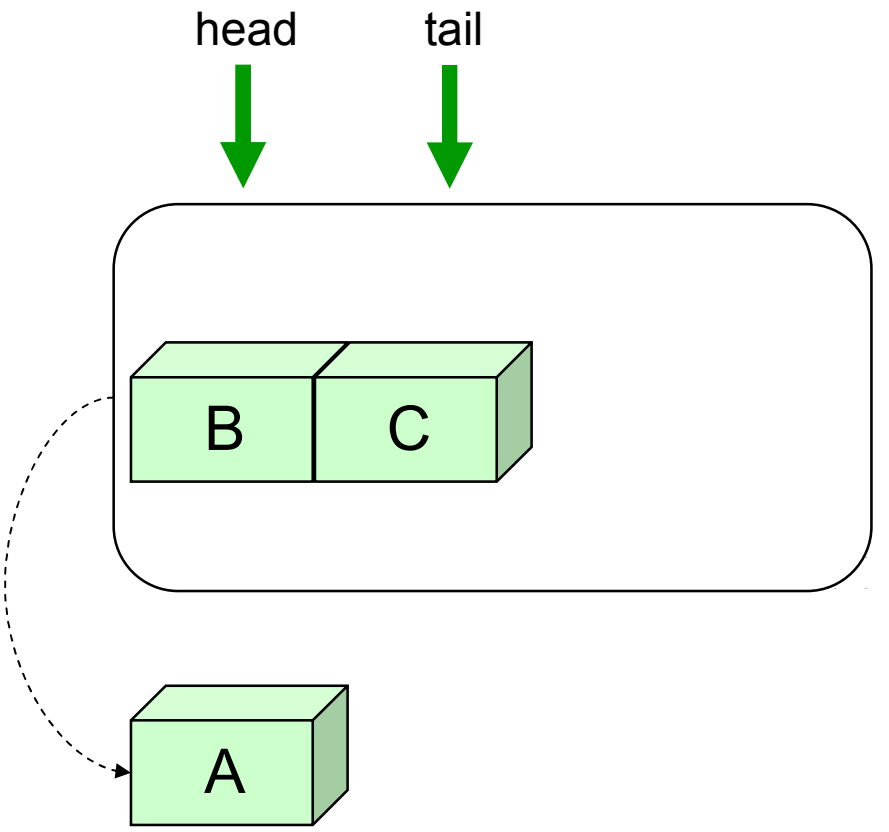
Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B
- inserir caixa C



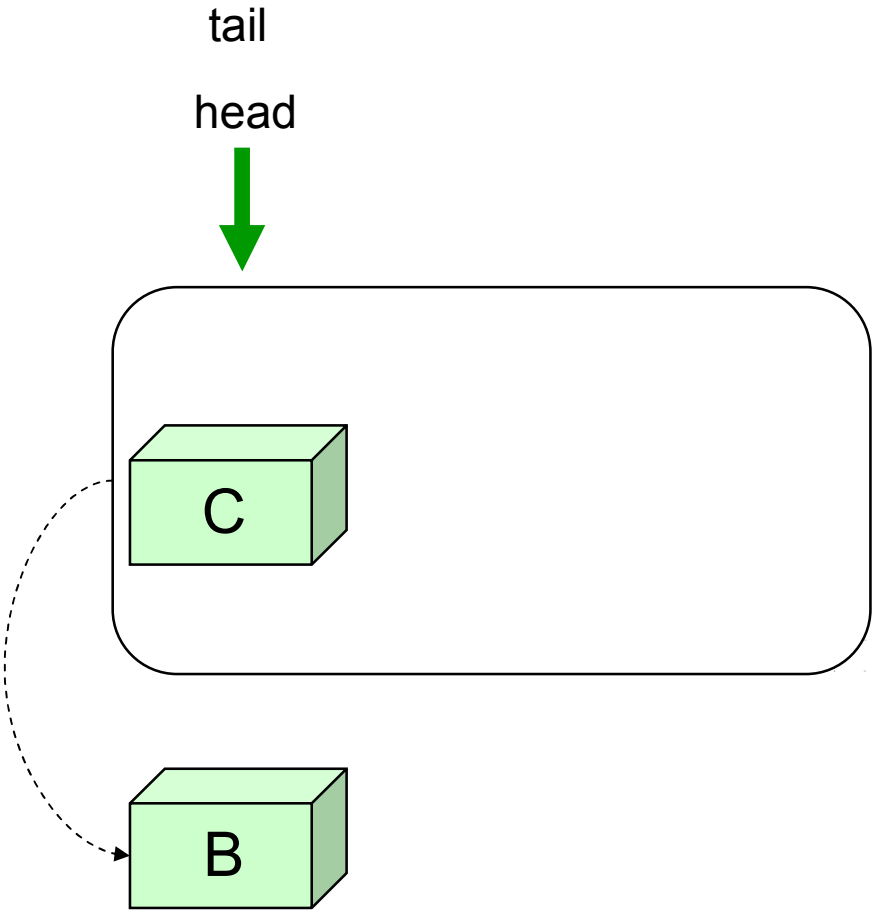
Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B
- inserir caixa C
- remover caixa



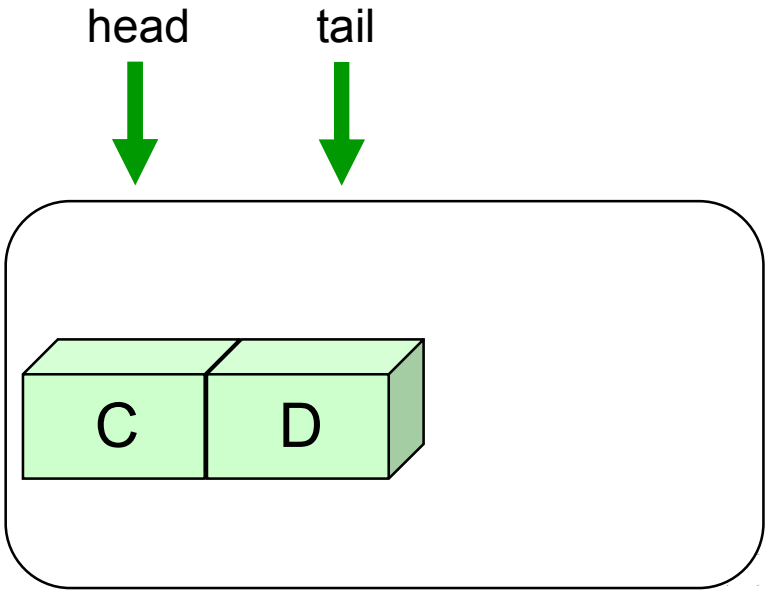
Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B
- inserir caixa C
- remover caixa
- remover caixa



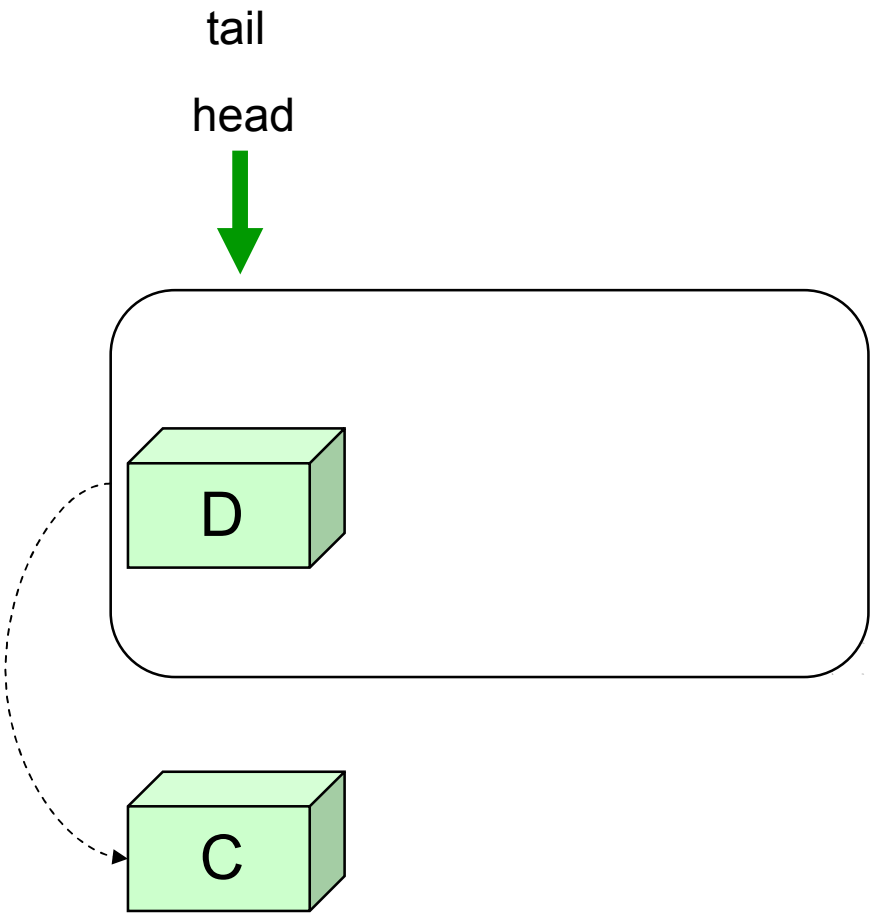
Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B
- inserir caixa C
- remover caixa
- remover caixa
- inserir caixa D



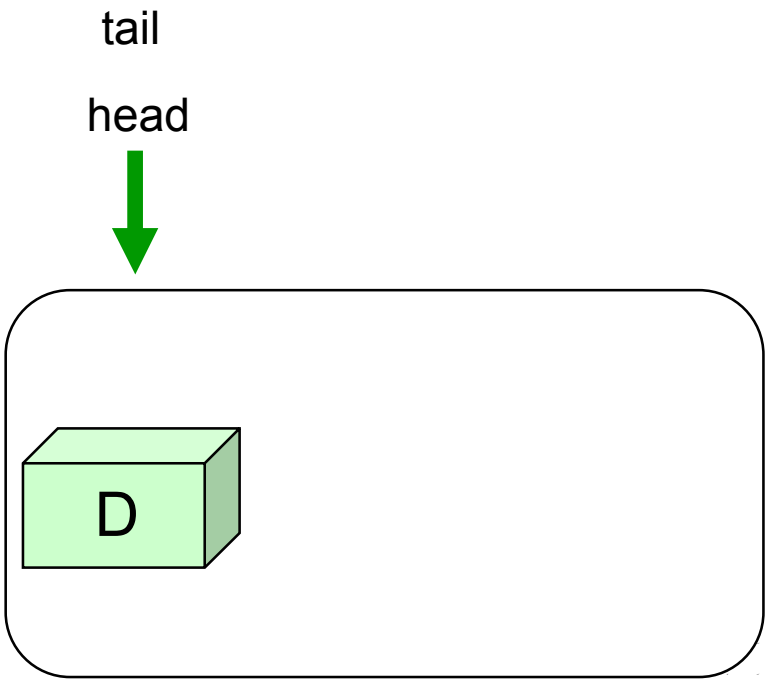
Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B
- inserir caixa C
- remover caixa
- remover caixa
- inserir caixa D
- remover caixa



Exemplo

- fila vazia inicialmente
- inserir caixa A
- inserir caixa B
- inserir caixa C
- remover caixa
- remover caixa
- inserir caixa D
- remover caixa



Estado final da fila

Especificação

- Operações:
  - Criação
  - Destruição
  - Status
  - Operações Básicas
  - Outras Operações

Criação

Queue::Queue();

- *pré-condição:* nenhuma
- *pós-condição:* Fila é criada e iniciada como vazia

Destruição

Queue::~Queue();

- *pré-condição:* Fila já tenha sido criada
- *pós-condição:* Fila é destruída, liberando espaço ocupado pelos seus elementos

Status

bool Queue::Empty();

- *pré-condição:* Fila já tenha sido criada
- *pós-condição:* função retorna **true** se a fila está vazia; **false** caso contrário

Status

bool Queue::Full();

- *pré-condição:* Fila já tenha sido criada
- *pós-condição:* função retorna **true** se a fila está cheia; **false** caso contrário

Operações Básicas

void Queue::Append(QueueEntry x);

- *pré-condição:* Fila já tenha sido criada e não está cheia
- *pós-condição:* O item **x** é armazenado no final da fila

O tipo **QueueEntry** depende da aplicação e pode variar desde um simples caracter ou número até uma **struct** ou **class** com muitos campos

Operações Básicas

void Queue::Serve(QueueEntry &x);

- *pré-condição:* Fila já tenha sido criada e não está vazia
- *pós-condição:* O item início da fila é removido e seu valor é retornado na variável **x**

Outras Operações

void Queue::Clear();

- *pré-condição:* Fila já tenha sido criada
- *pós-condição:* Todos os itens da fila são descartados e ela torna-se uma fila vazia

Outras Operações

int Queue::Size();

- *pré-condição:* Fila já tenha sido criada
- *pós-condição:* função retorna o número de itens na fila

Outras Operações

void Queue::Front(QueueEntry &x);

- *pré-condição:* fila já tenha sido criada e não está vazia
- *pós-condição:* A variável **x** recebe uma cópia do item que encontra-se no início da fila; a fila permanece inalterada

Outras Operações

void Queue::Rear(QueueEntry &x);

- *pré-condição:* fila já tenha sido criada e não está vazia
- *pós-condição:* A variável **x** recebe uma cópia do item que encontra-se no final da fila; a fila permanece inalterada

Pontos Importantes

- Uma analogia útil para o ADT Fila consiste em pensar em uma fila de pessoas no banco ou no supermercado:
  - O primeiro cliente que chega na fila é o primeiro a ser atendido
- As operações **Append** e **Serve** são também conhecidas como **Insert** e **Delete** ou **Enqueue** e **Dequeue**, respectivamente

Implementação Contígua

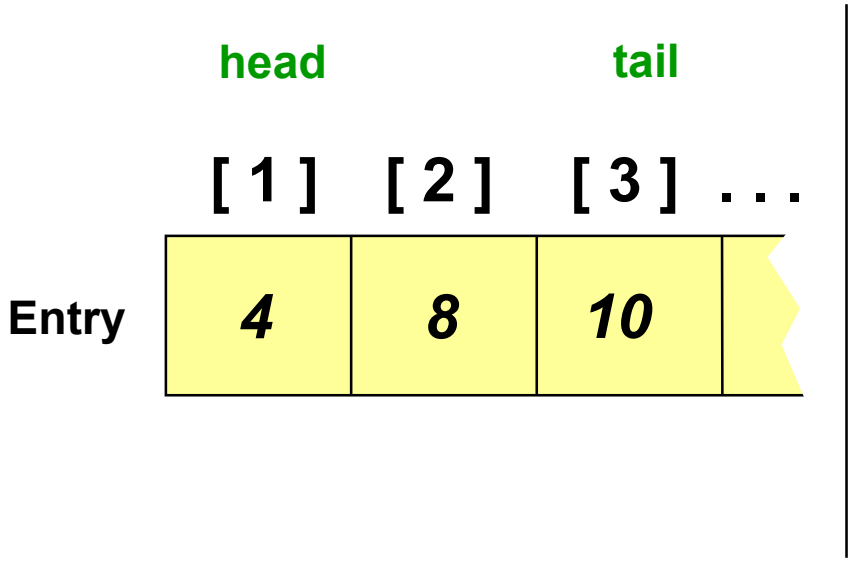
- Como no caso de pilhas, veremos inicialmente os detalhes de implementação de filas utilizando vetores (*arrays*)
- Logo a seguir, veremos a implementação encadeada dinâmica de filas

Possíveis Implementações Contíguas

- Modelo físico
  - Um vetor linear com a frente (**head**) sempre na primeira posição e todas as entradas são movidas no vetor quando um item é removido
  - Geralmente, este é um método lento e pobre para ser usado em computadores
- Vetor linear com dois índices
  - frente (**head**) e final (**tail**) que sempre crescem
  - É um bom método se a fila pode ser esvaziada totalmente quando cheia
- Vetor circular

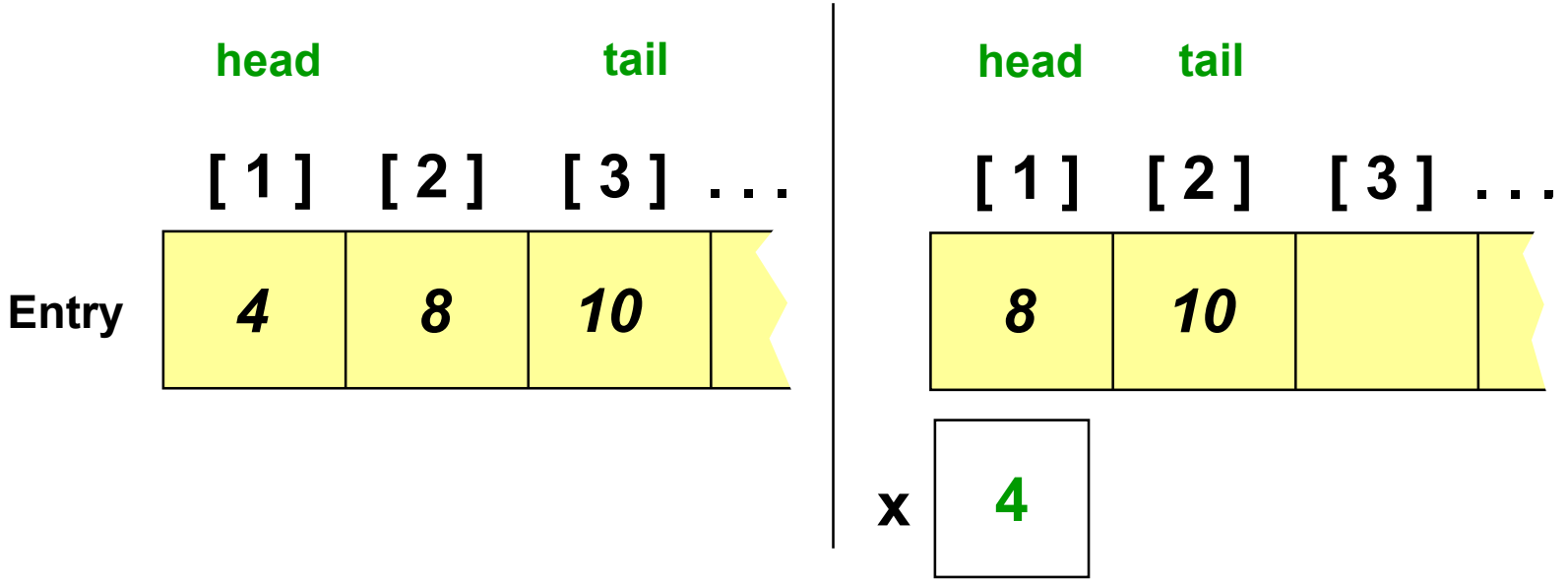
Modelo Físico

Antes da chamada a Serve(x)
nós temos esta fila:



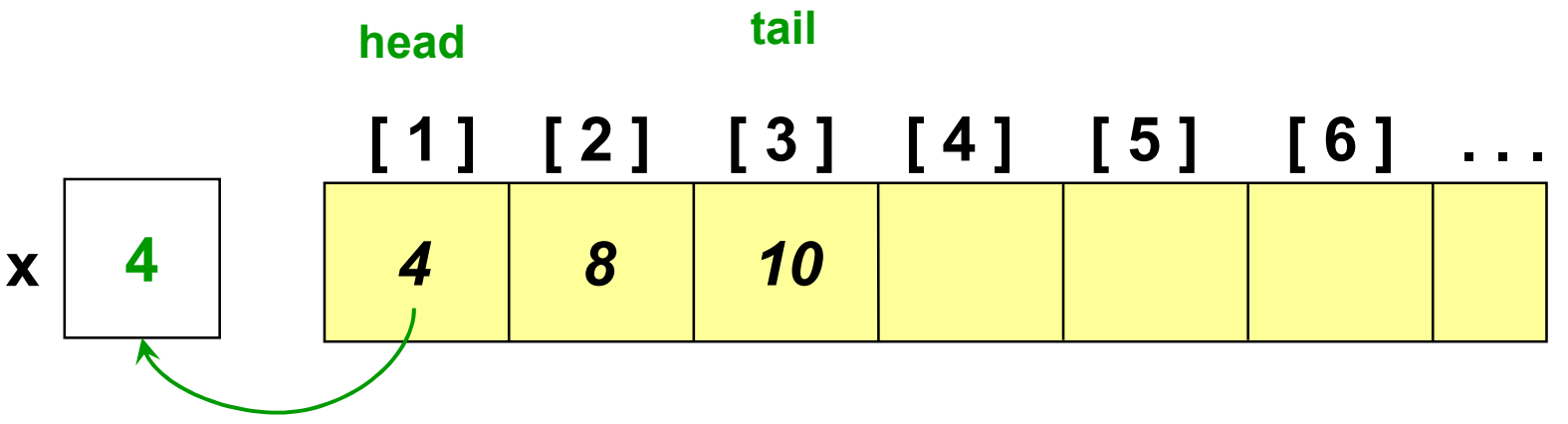
Modelo Físico

Depois da chamada a Serve(x) nós teremos esta fila:



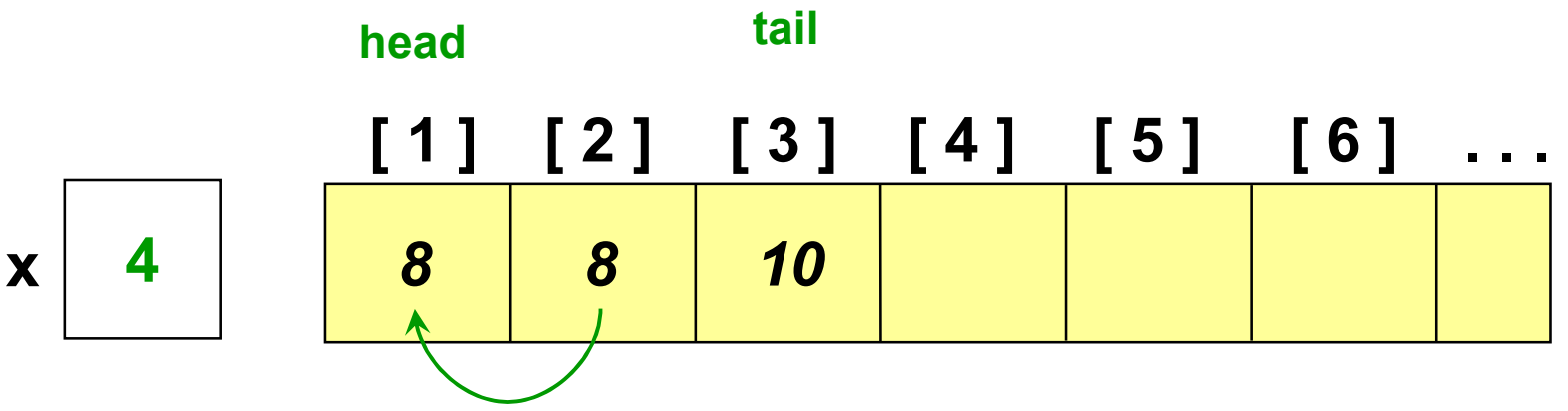
Modelo Físico

- Fila após ativar Serve(x)...



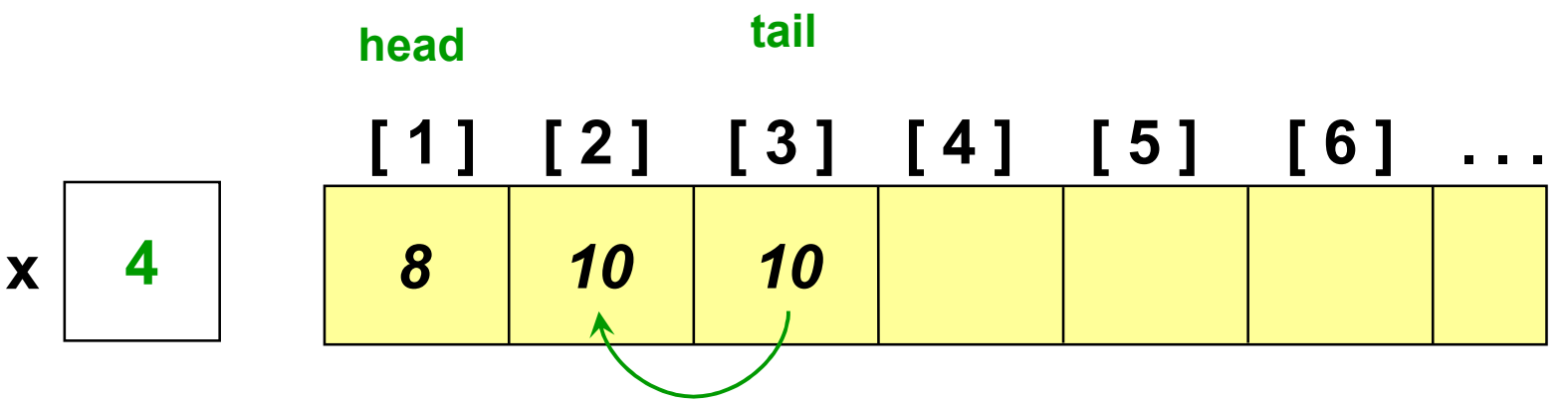
Modelo Físico

- Fila após ativar Serve(x)...



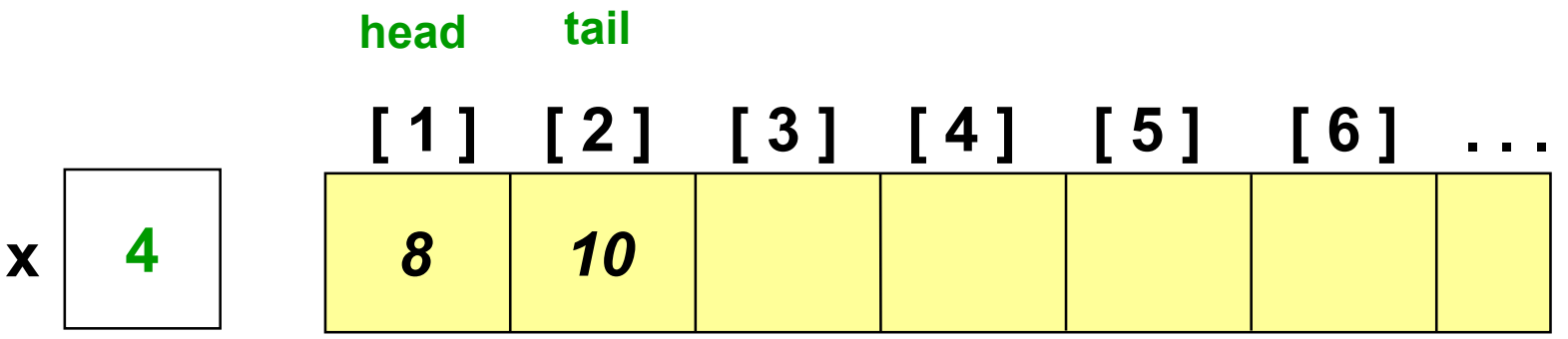
Modelo Físico

- Fila após ativar Serve(x)...



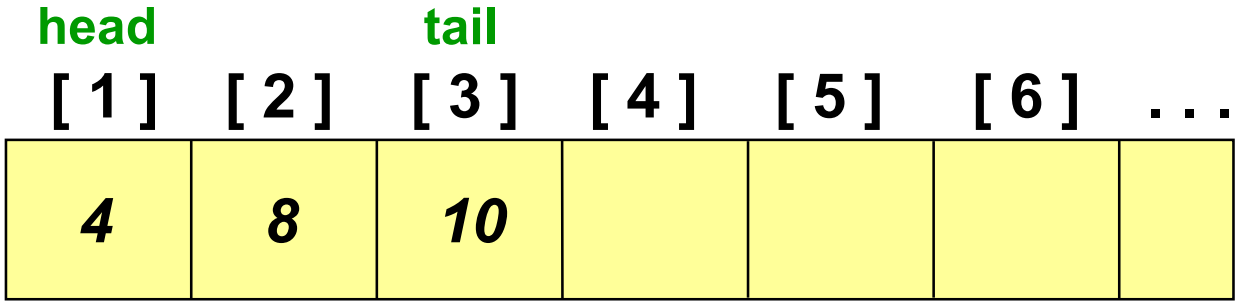
Modelo Físico

- Fila após ativar Serve(x)...



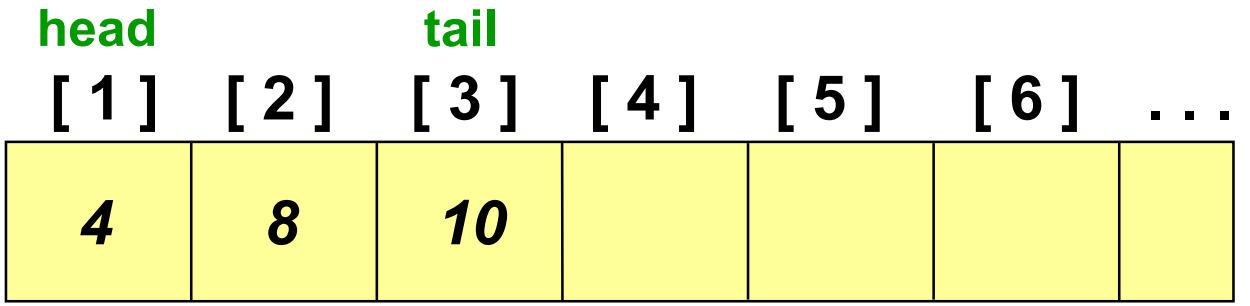
Vetor com Dois Índices

- Fila antes de ativar Serve(x)...

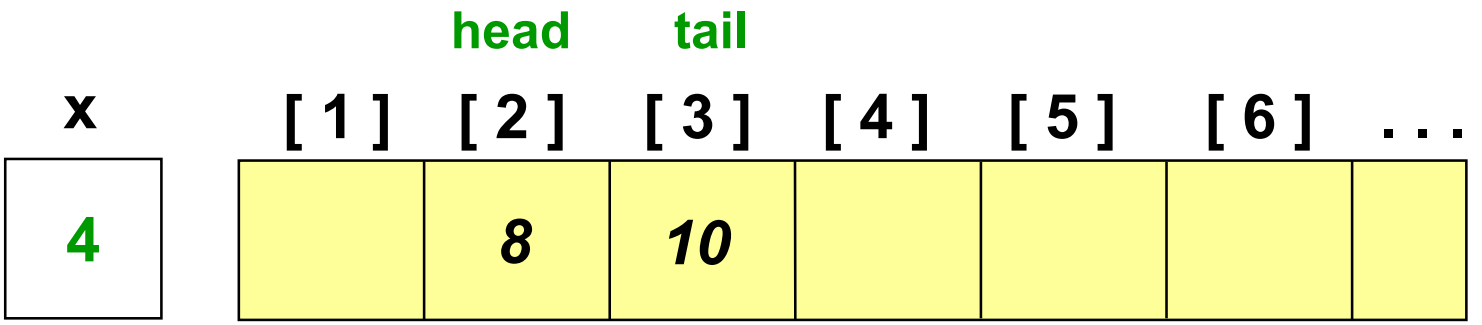


Vetor com Dois Índices

- Fila antes de ativar Serve(x)...



- Fila após ativar Serve(x)...

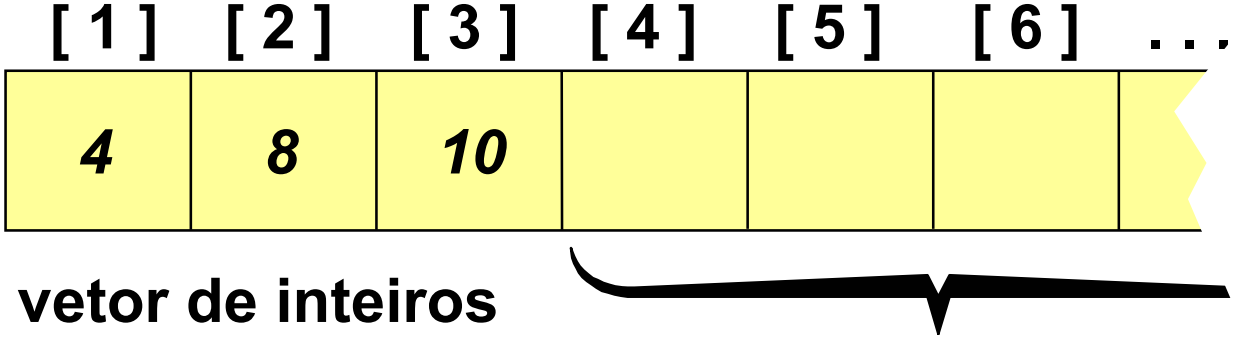


Vetor Circular

- Para implementar um vetor circular em um vetor linear, suponha um círculo numerado de 1 até **max**, onde **max** indica o número de elementos do vetor circular
- O movimento de índices é dado pelo módulo aritmético: quando um índice ultrapassa **max**, ele recomeça novamente em 1
- Isto é similar a um relógio, onde as horas são numeradas de 1 a 12 e ao adicionar 4 horas às 10 horas, obtém-se 2 horas

Detalhes de Implementação

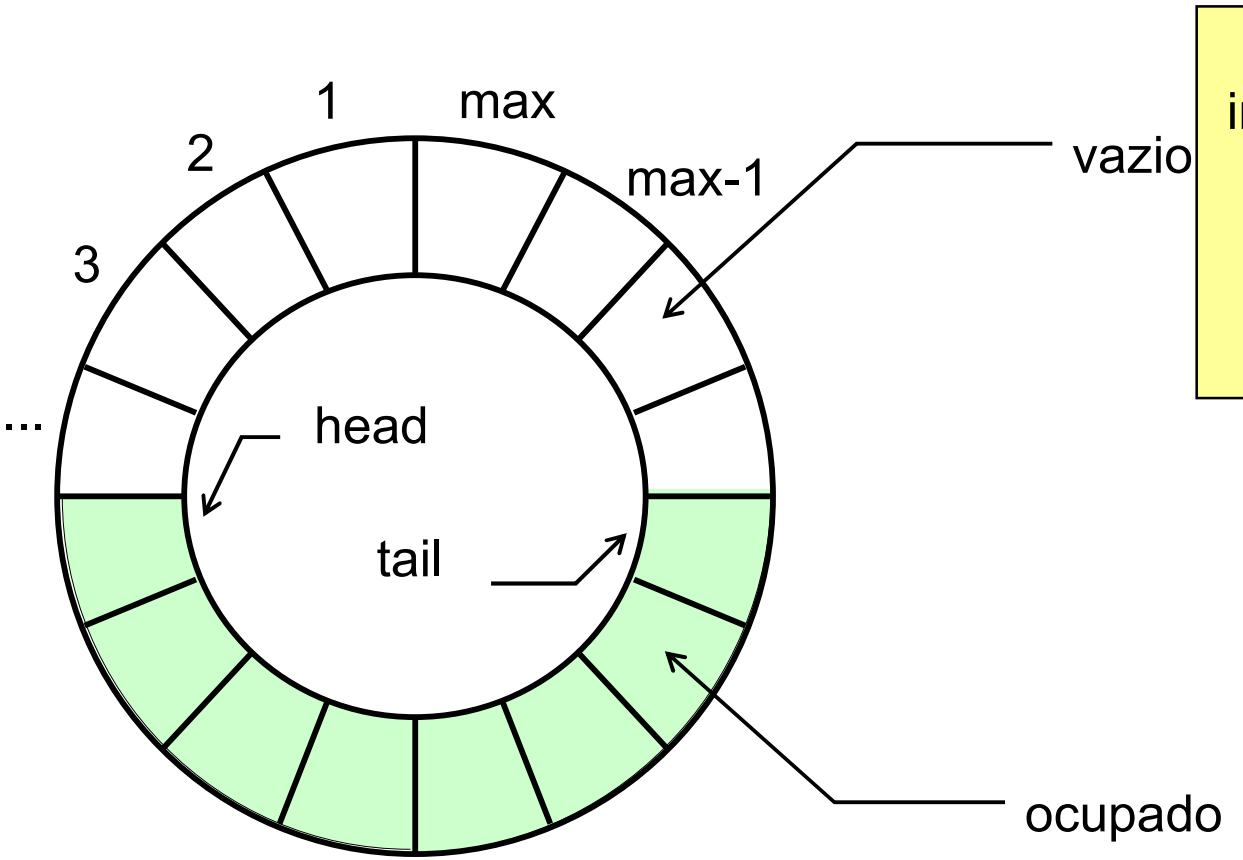
- As entradas em uma Fila serão inicialmente armazenadas no **início de um vetor**, como mostra este exemplo



Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor

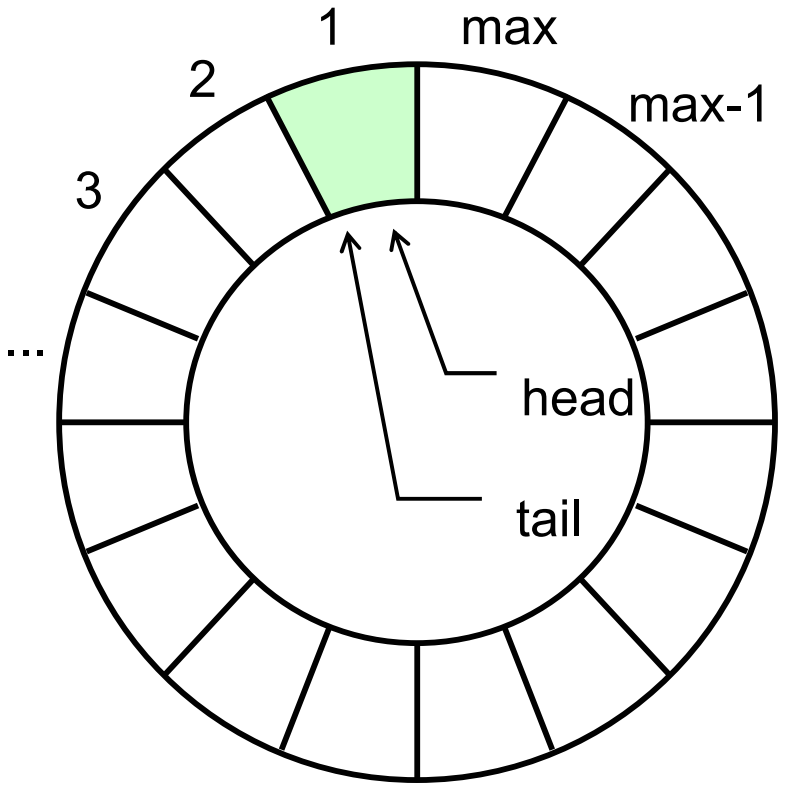
Detalhes de Implementação



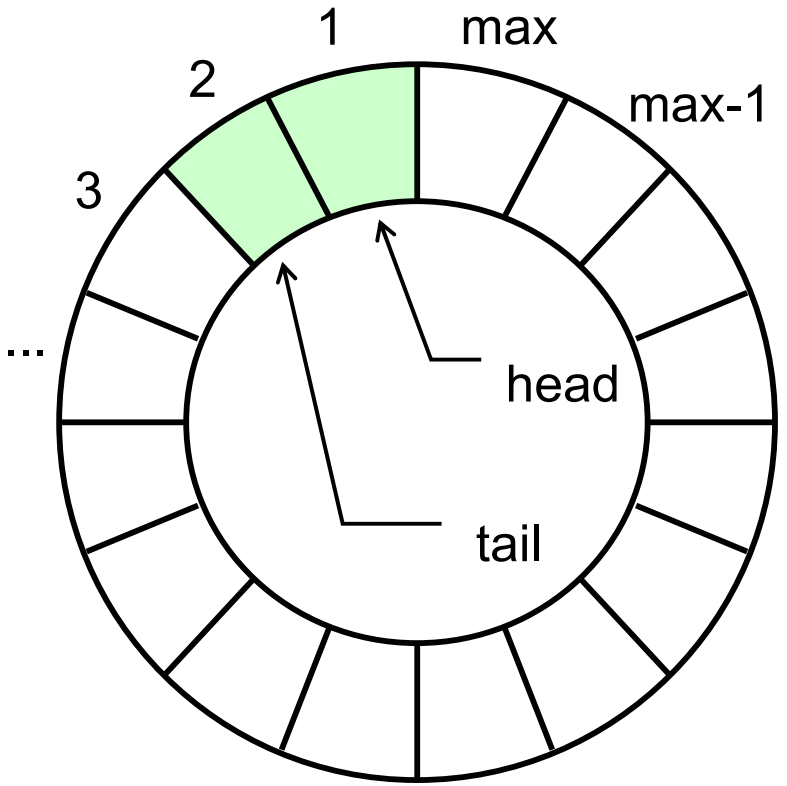
Nós não nos interessamos para o que está armazenado nesta parte do vetor

Exemplo

- Inserindo um item

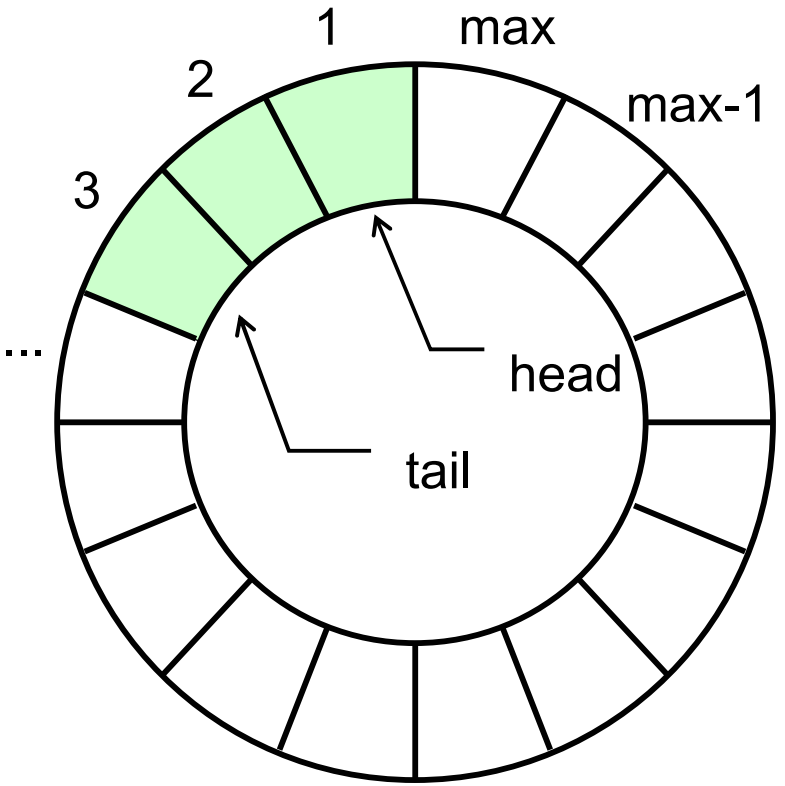


Exemplo



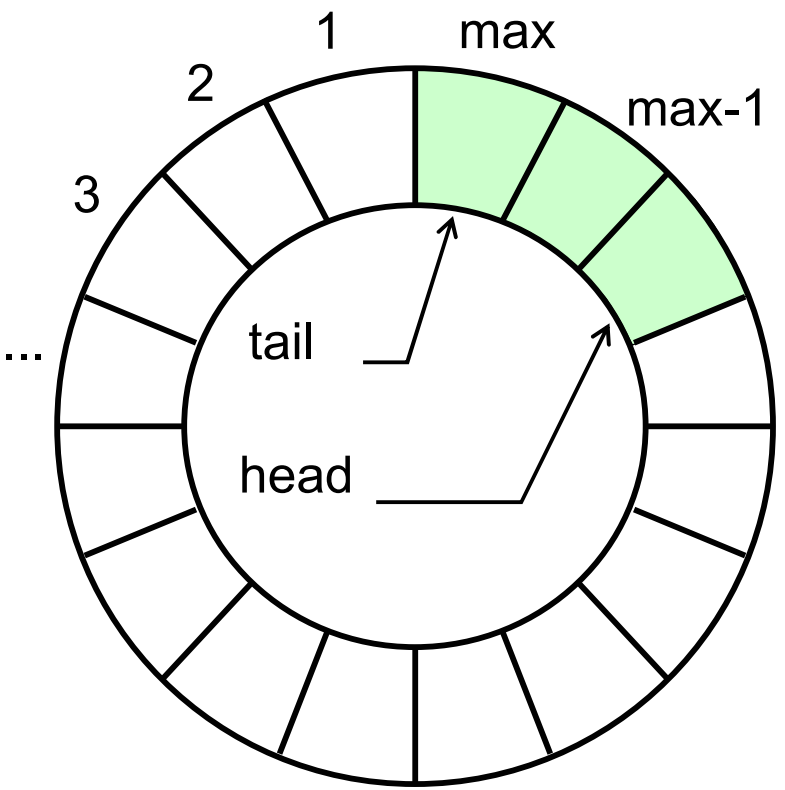
- Inserindo um item...
- mais um...

Exemplo



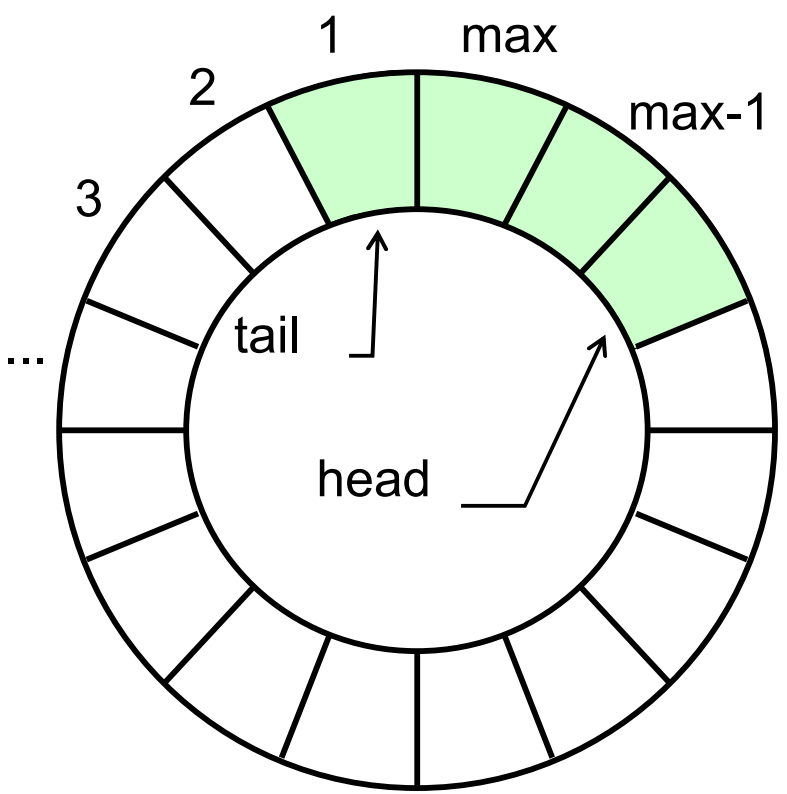
- Inserindo um item...
- mais um...
- e mais um...

Exemplo



- Depois de várias inserções e remoções, chega-se ao final do vetor...

Exemplo



- Depois de várias inserções e remoções, chega-se ao final do vetor...
- ... e recomeça-se da posição 1

Incremento no Vetor Circular

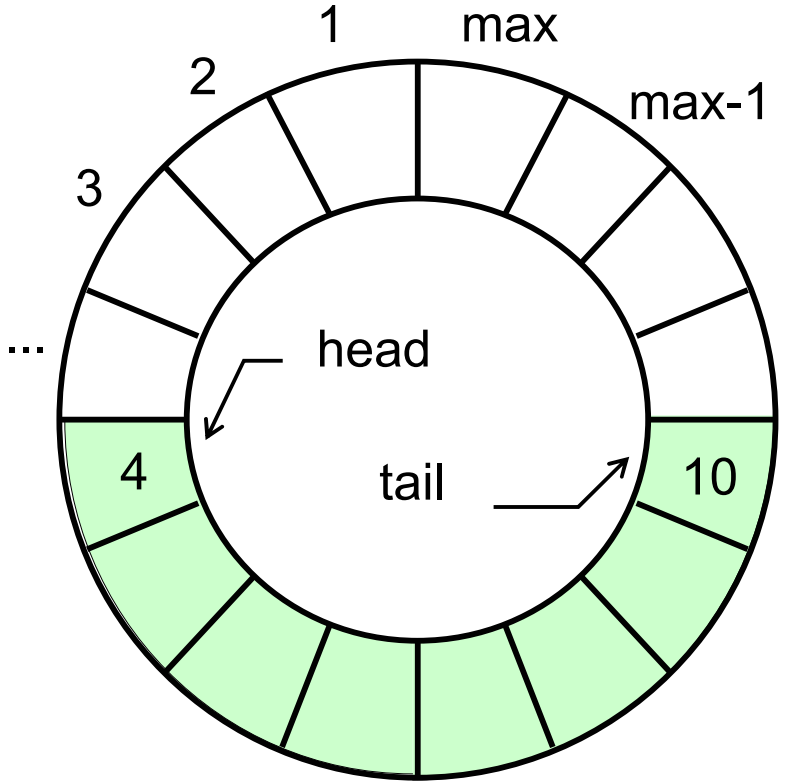
- Em C++ o incremento unitário de um índice **i** pode ser escrito como:

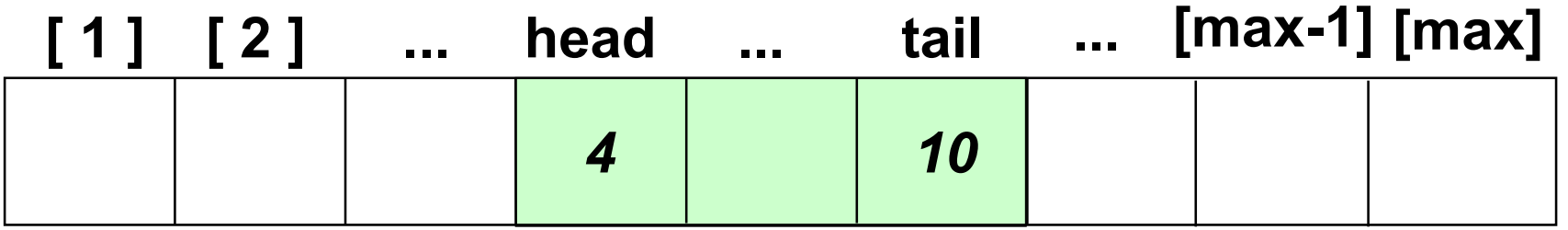
```
if (i == max)
  i = 1;
else
  i++;
```

- ...que é equivalente a:

```
i = (i % max) + 1;
```

“Desenrolando” o Vetor...





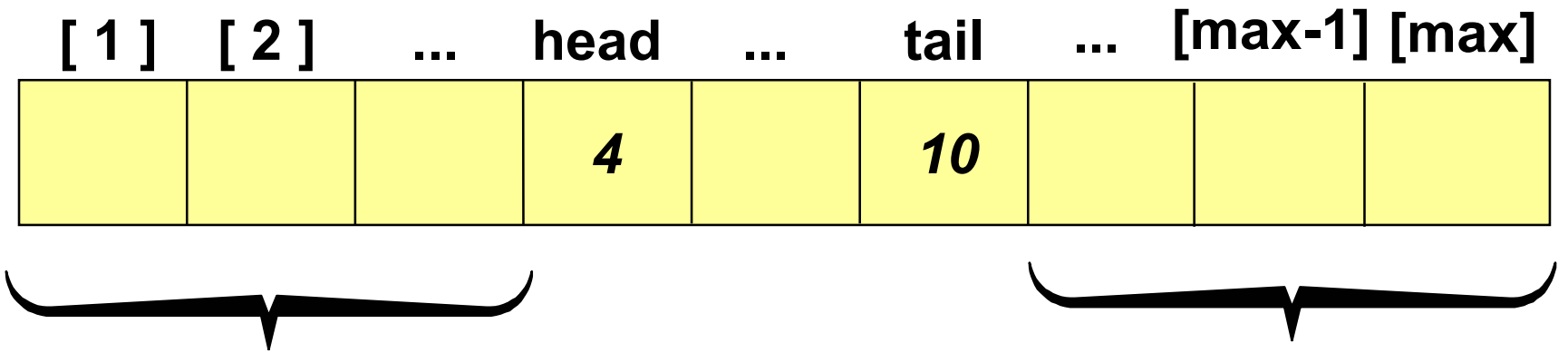
Vetor Circular
Possíveis Implementações

- Vetor circular com dois índices (**head** e **tail**) e uma posição sempre deixada vaga
- Vetor circular com dois índices (**head** e **tail**) e uma variável inteira (**count**) contendo o número de itens
- Vetor circular com dois índices (**head** e **tail**) que assumem valores especiais para indicar fila vazia

Detalhes de Implementação

- Nós precisamos armazenar os elementos da fila...

Um vetor de inteiros

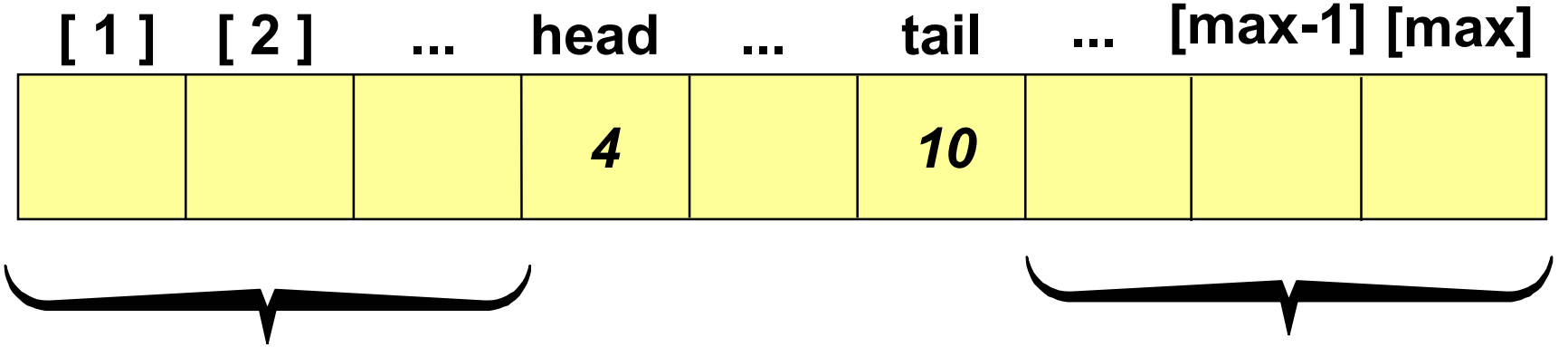


Nós não nos interessamos para o que está armazenado nestas partes do vetor

Detalhes de Implementação

- ... bem como inteiros que indicam o início, final e contador de elementos da fila:

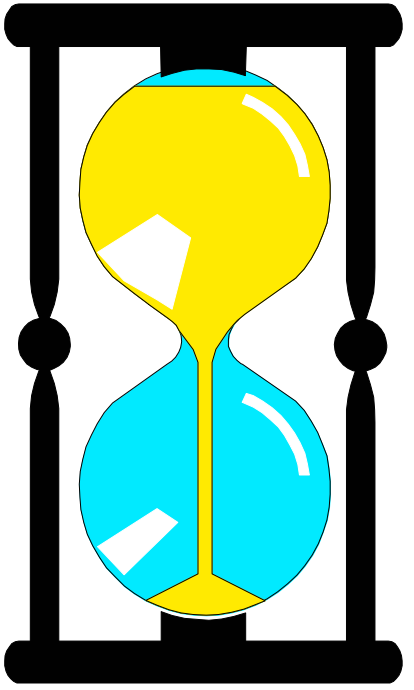
Um vetor de inteiros



Nós não nos interessamos para o que está armazenado nestas partes do vetor

Questão

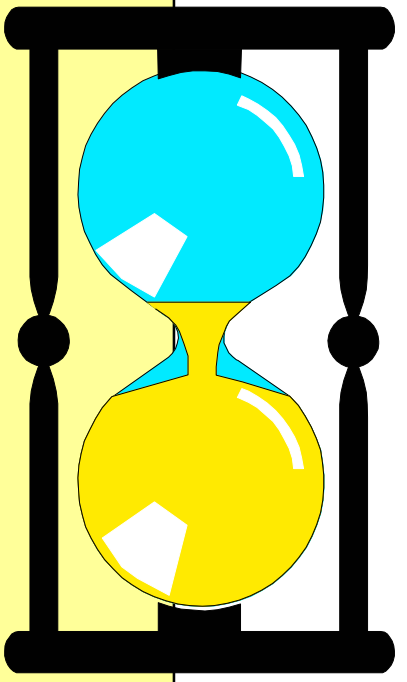
- Utilize estas idéias para escrever uma declaração de tipo que poderia implementar o tipo de dado fila. A declaração deve ser um objeto com quatro campos de dados. Faça uma fila capaz de armazenar 100 inteiros



Você tem 3 minutos para escrever a declaração

Uma Solução

```
const int MaxQueue = 100;
class Queue
{ public:
     Queue();                  // construtor
     void Append(int x);
     void Serve(int &x);
     ...
  private:
    int head;                  // início da fila
    int tail;                  // final da fila
    int count;                 // nº. de elementos fila
    int Entry[MaxQueue+1];  // vetor com elementos
};
```



Uma Solução

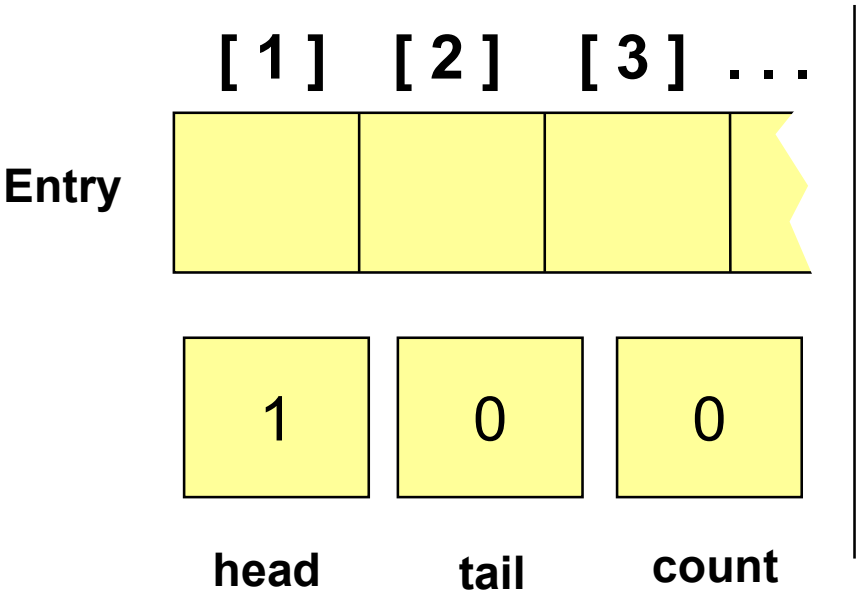
```
const int MaxQueue = 100;
class Queue
{ public:
     Queue();                    // construtor
     void Append(int x);
     void Serve(int &x);
     ...
  private:
    int head;                    // início da fila
    int tail;                    // final da fila
    int count;                   // nº. de elementos fila
    int Entry[MaxQueue+1];  // vetor com elementos
};
```

Observe que o tipo **QueueEntry** nesse caso é um inteiro

Construtor

Queue::Queue()

Numa fila vazia, **tail** fica uma posição antes de **head**

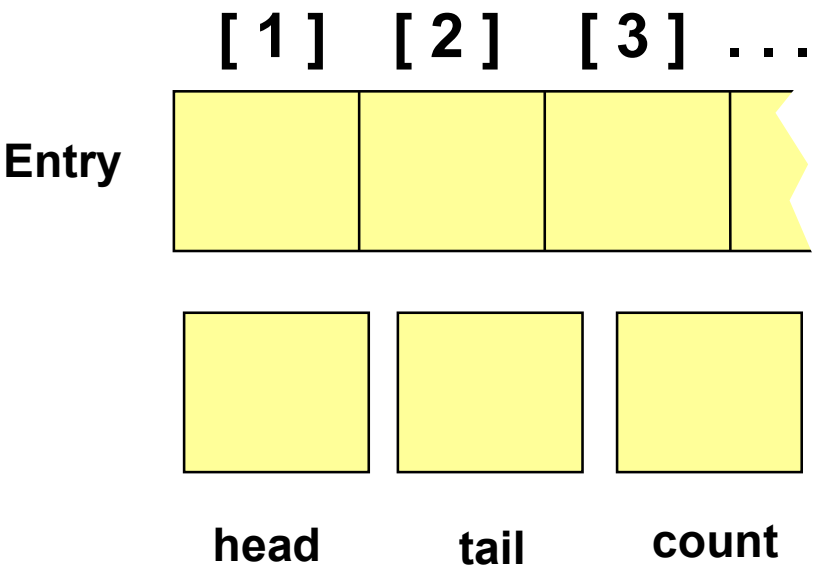



```
Queue::Queue()
{  count = 0;
   head = 1;
   tail = 0;
}
```

Destruidor

Queue::~Queue()

Usando alocação estática para implementar a fila (vetor), o destruidor não será necessário. Em todo caso, colocaremos apenas uma mensagem que o objeto foi destruído

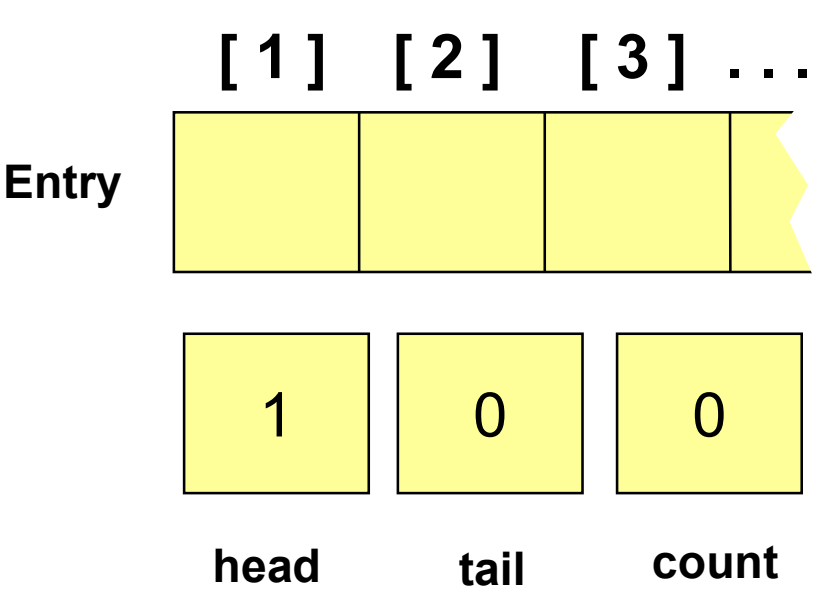



```
Queue::~Queue()
{
    cout << “Fila destruída”;
}
```

Status: Empty

bool Queue::Empty()

Lembre-se que a fila possui um contador de elementos...

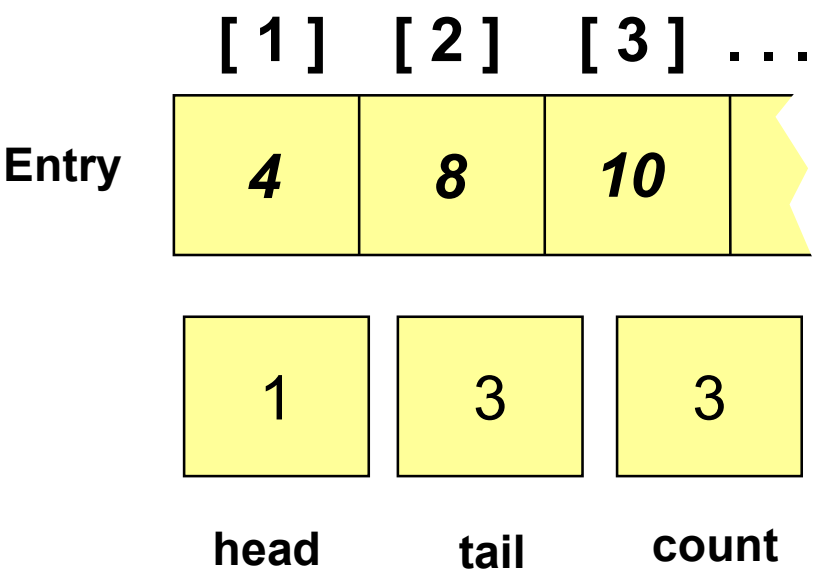



```
bool Queue::Empty()
{
    return (count == 0);
}
```

Status: Full

bool Queue::Full()

... e que **MaxQueue** é o número máximo de elementos da fila.

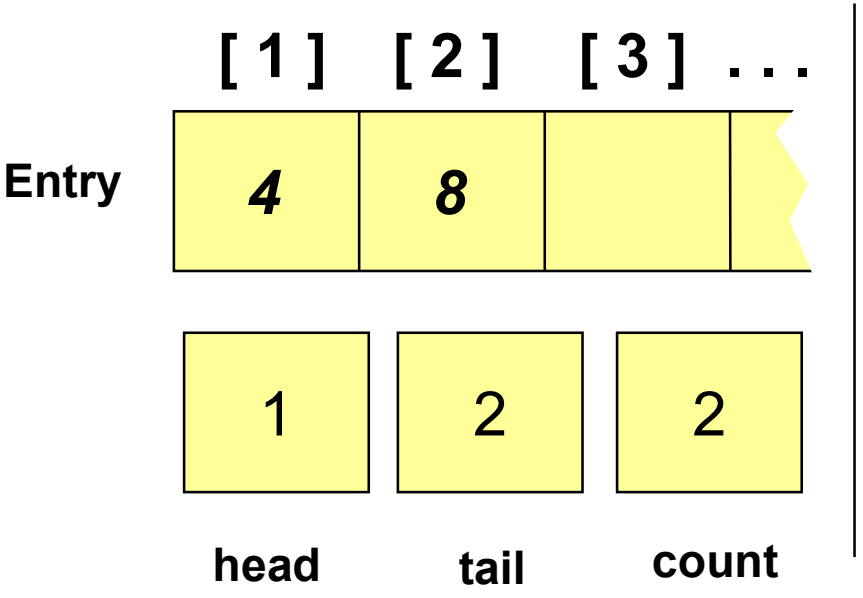



```
bool Queue::Full()
{
    return (count == MaxQueue);
}
```

Operações Básicas: Append

```
void Queue::Append(int x)
```

Nós fazemos uma chamada a
Append(10)

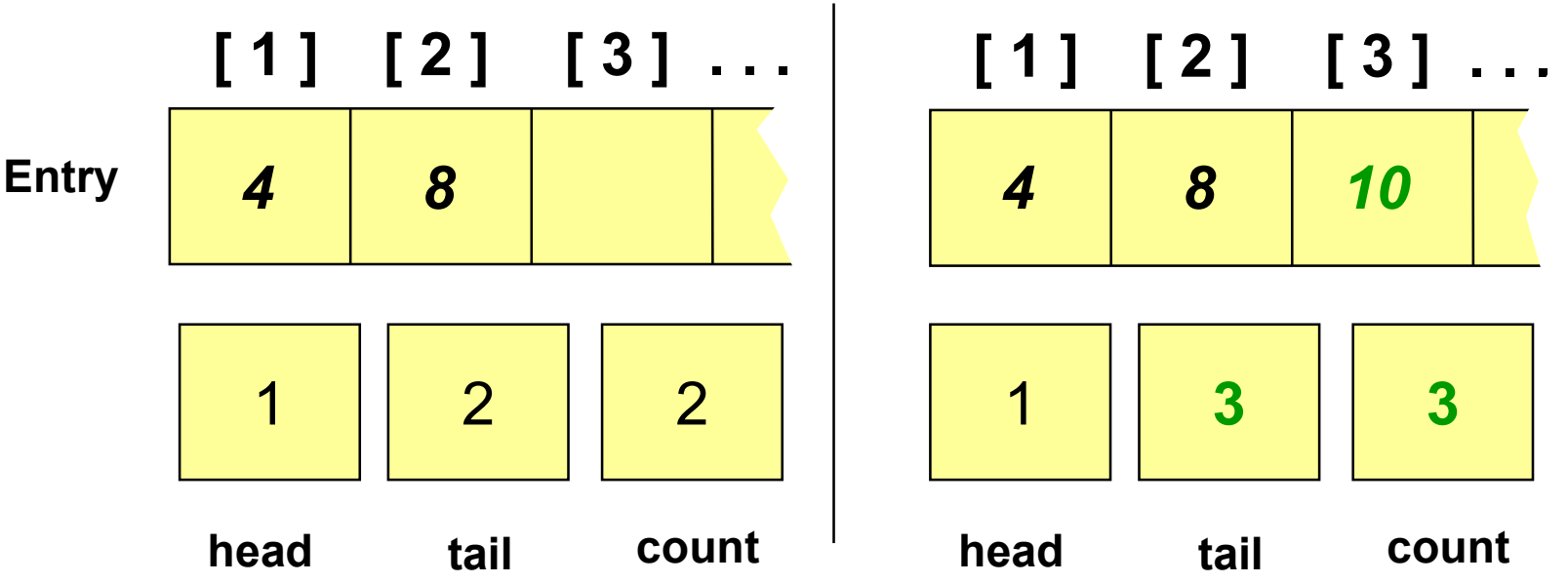


Quais valores serão armazenados em **Entry**, **head**, **tail** e **count** depois que a chamada de procedimento termina?

Operações Básicas: Append

```
void Queue::Append(int x)
```

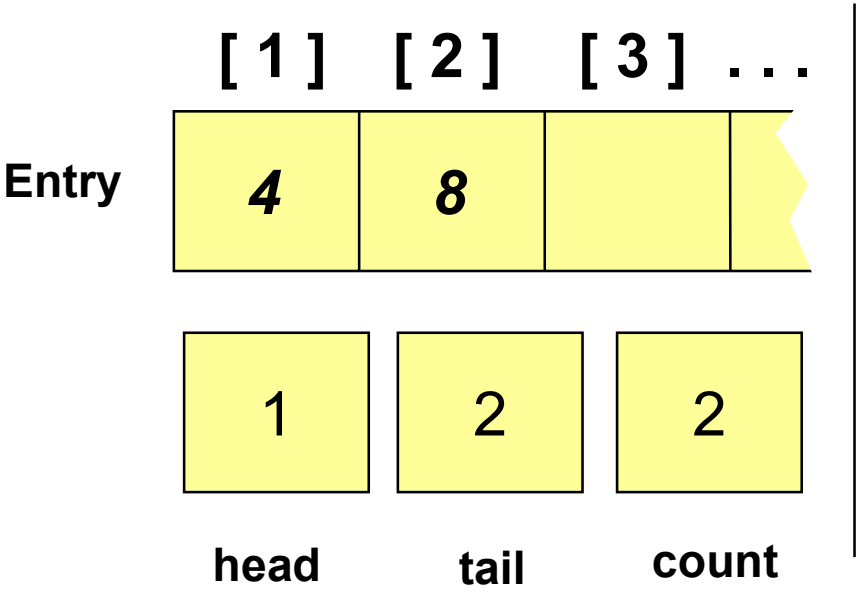
Depois da chamada a Append(10), nós teremos esta fila:



Operações Básicas: Append

void Queue::Append(int x)

Antes de inserir, é conveniente verificar se há espaço na fila

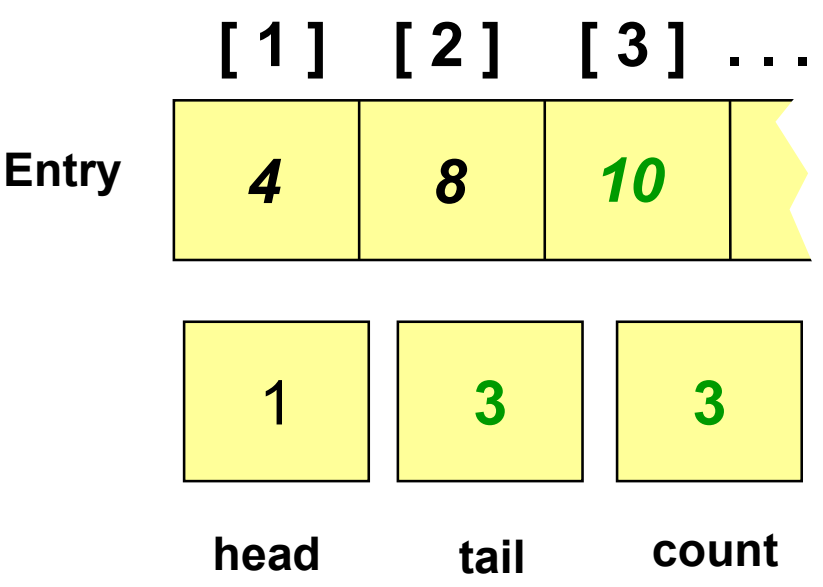



```
void Queue::Append(int x)
{  if (Full())
   {  cout << “Fila Cheia”;
      abort();
   }
  ...
```

Operações Básicas: Append

void Queue::Append(int x)

Se houver, basta inserir na próxima posição livre do vetor

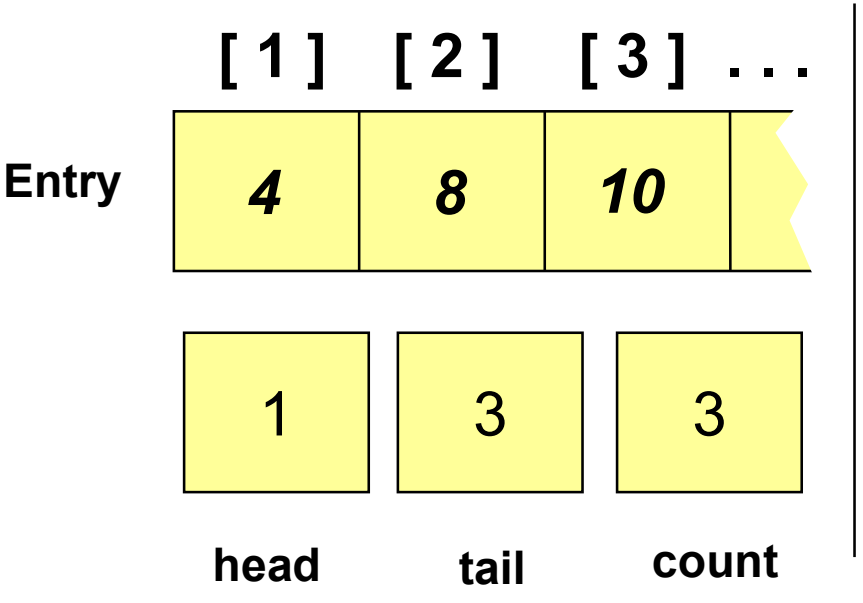



```
void Queue::Append(int x)
{  if (Full())
   {  cout << “Fila Cheia”;
      abort();
   }

   count++;
   tail = (tail % MaxQueue)+1;
   Entry[tail] = x;
}
```

Operações Básicas: Serve

```
void Queue::Serve(int &x)
```

Nós fazemos uma chamada a Serve(x)

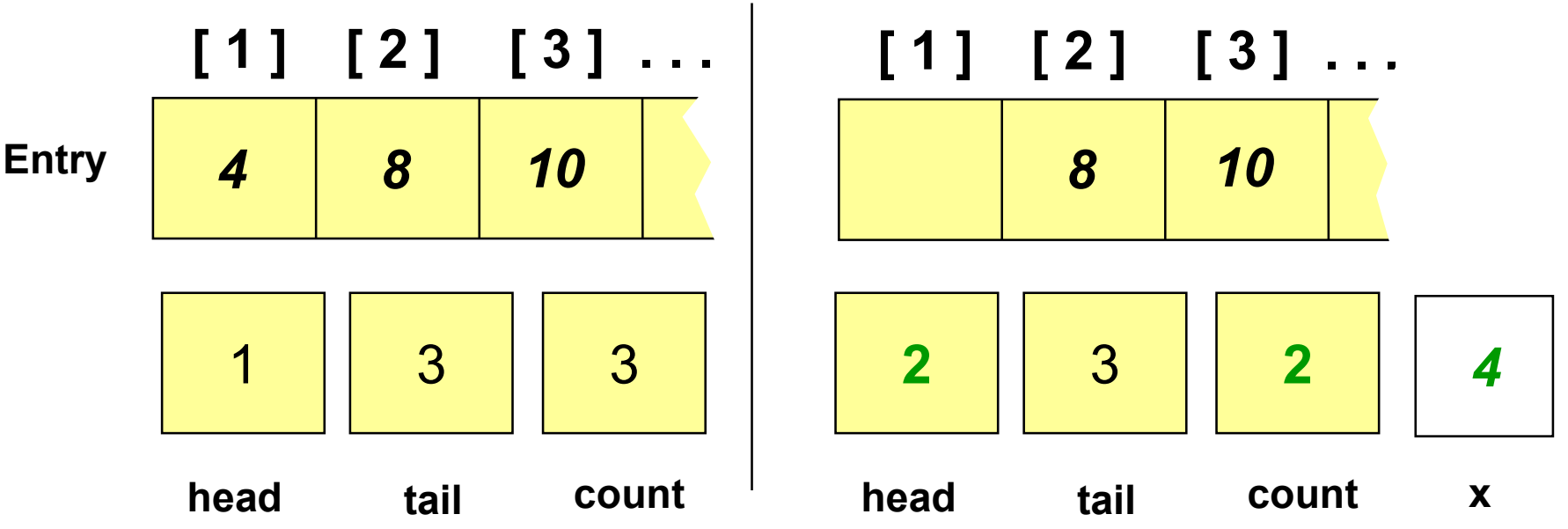


Quais valores serão armazenados em **Entry**, **head**, **tail** e **count** depois que a chamada de procedimento termina?

Operações Básicas: Serve

```
void Queue::Serve(int &x)
```

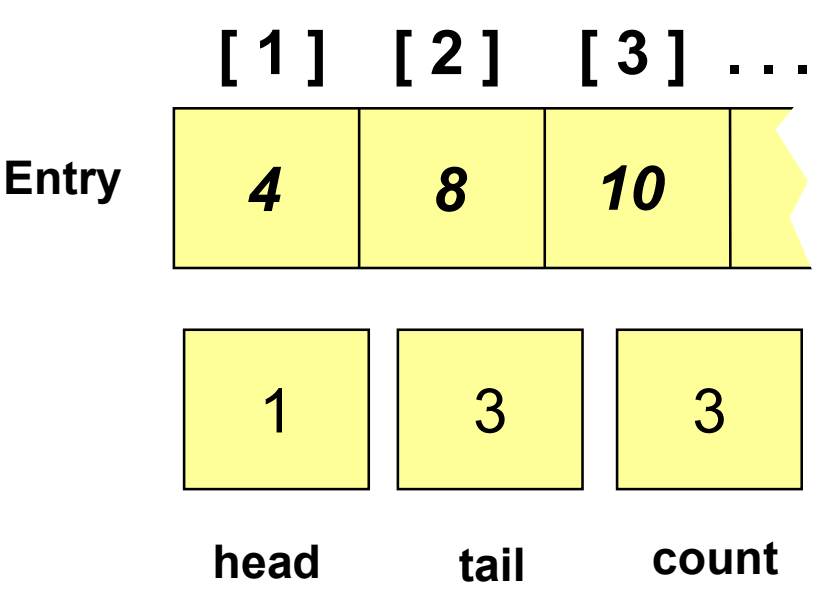
Depois da chamada a Serve(x), nós teremos esta fila:



Operações Básicas: Serve

void Queue::Serve(int &x)

Antes de remover, é conveniente verificar se a fila não está vazia

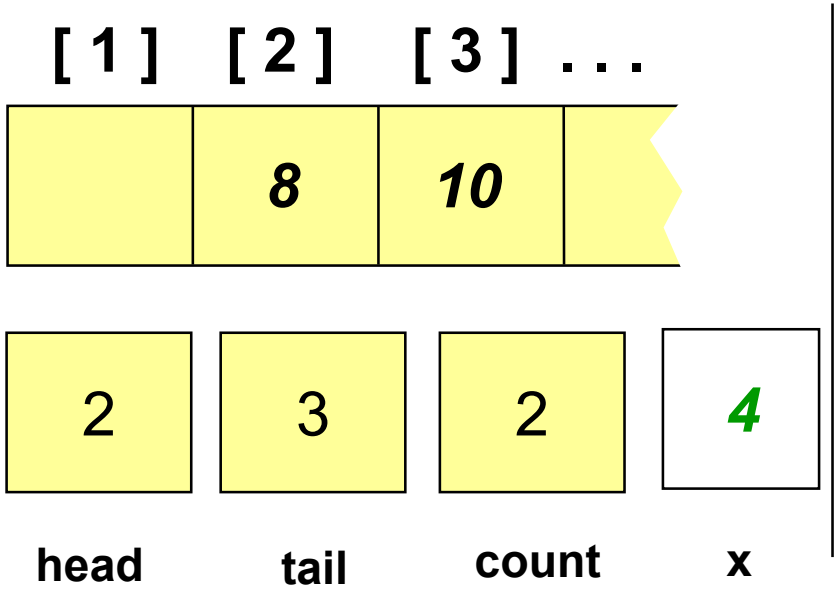



```
void Queue::Serve(int &x)
{  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
   ...
```

Operações Básicas: Serve

void Queue::Serve(int &x)

Se não estiver vazia, basta retirar o elemento apontado por **head**:




```
void Queue::Serve(int &x)
{  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }

   count = count - 1;
   x = Entry[head] ;
   head=(head % MaxQueue)+1;
}
```

Exercícios

- Implemente Clear(), usando apenas Serve() e Empty()
- Implemente Clear() utilizando campos do objeto
- Implemente Front() e Rear()
- Implemente Size()

Solução Clear

- Usando apenas Serve e Empty

```
void Queue::Clear()
{ int x;

  while(! Empty())
    Serve(x);
}
```

- Utilizando campos do objeto

```
void Queue::Clear()
{  count = 0;
   head = 1;
   tail = 0;
}
```

Solução Front/Rear

```
void Queue::Front(int &x)
{  if(Empty())
   { cout << “Fila vazia”;
     abort();
   }
   x = Entry[head];
}
```

```
void Queue::Rear(int &x)
{  if(Empty())
   { cout << “Fila vazia”;
     abort();
   }
   x = Entry[tail];
}
```

Solução Size

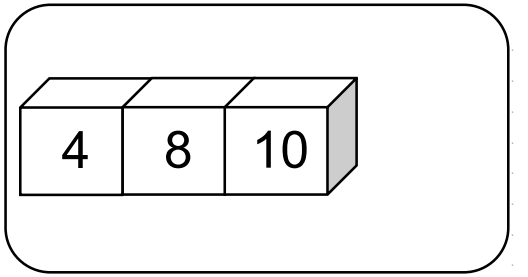
```
int Queue::Size()
{
   return count;
}
```

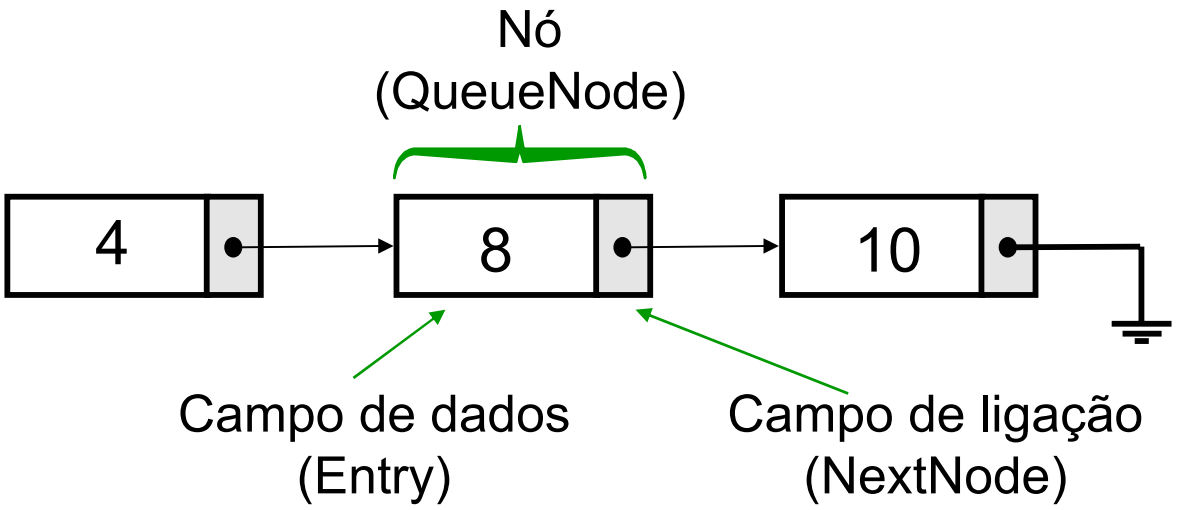
Pontos Importantes

- Até este ponto vimos uma forma de implementação de filas usando vetores circulares
- A vantagem de usar vetores circulares é a simplicidade dos programas e aproveitamento melhor do espaço alocado
- Nos próximos *slides* veremos a implementação utilizando alocação dinâmica de memória

Detalhes de Implementação

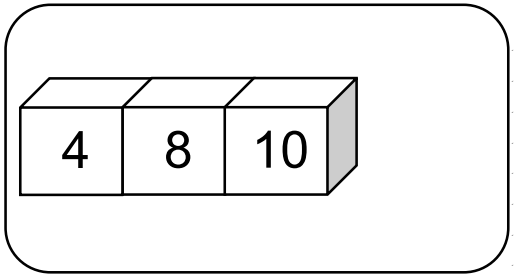
- Assim como na implementação de pilhas, as entradas de uma fila são colocadas em um estrutura (**QueueNode**) que contém um campo com o valor existente na fila (**Entry**) e outro campo é um apontador para o próximo elemento na fila (**NextNode**)



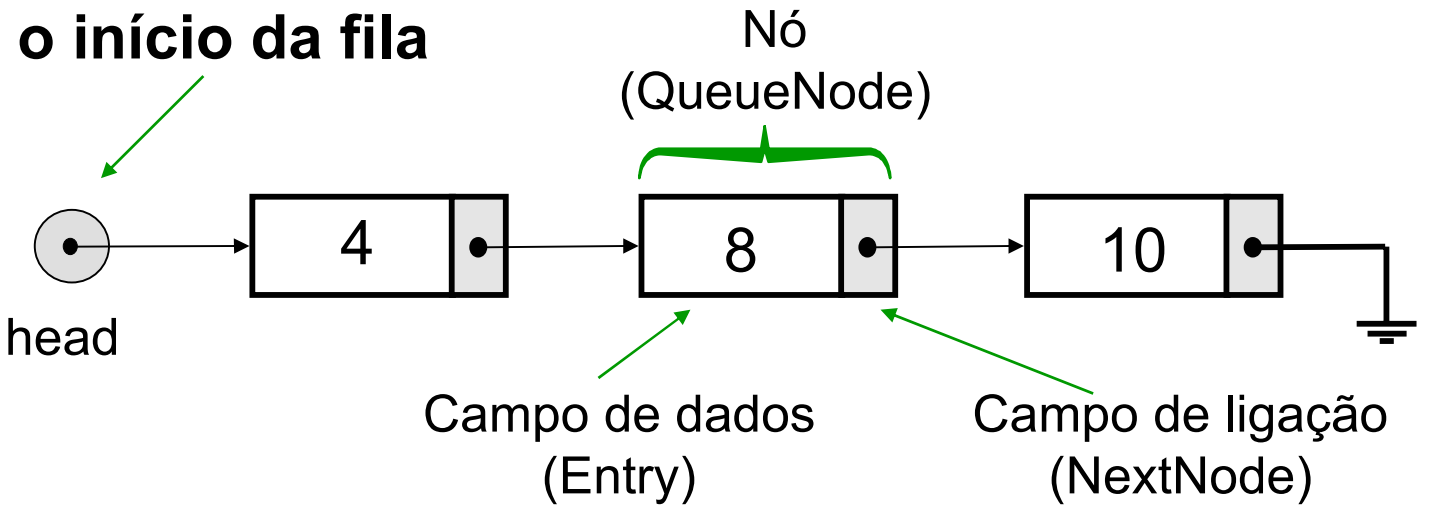


Detalhes de Implementação

- Nós precisamos armazenar o início e final da fila

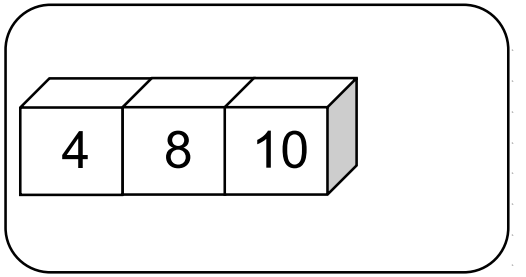


Um ponteiro armazena o início da fila

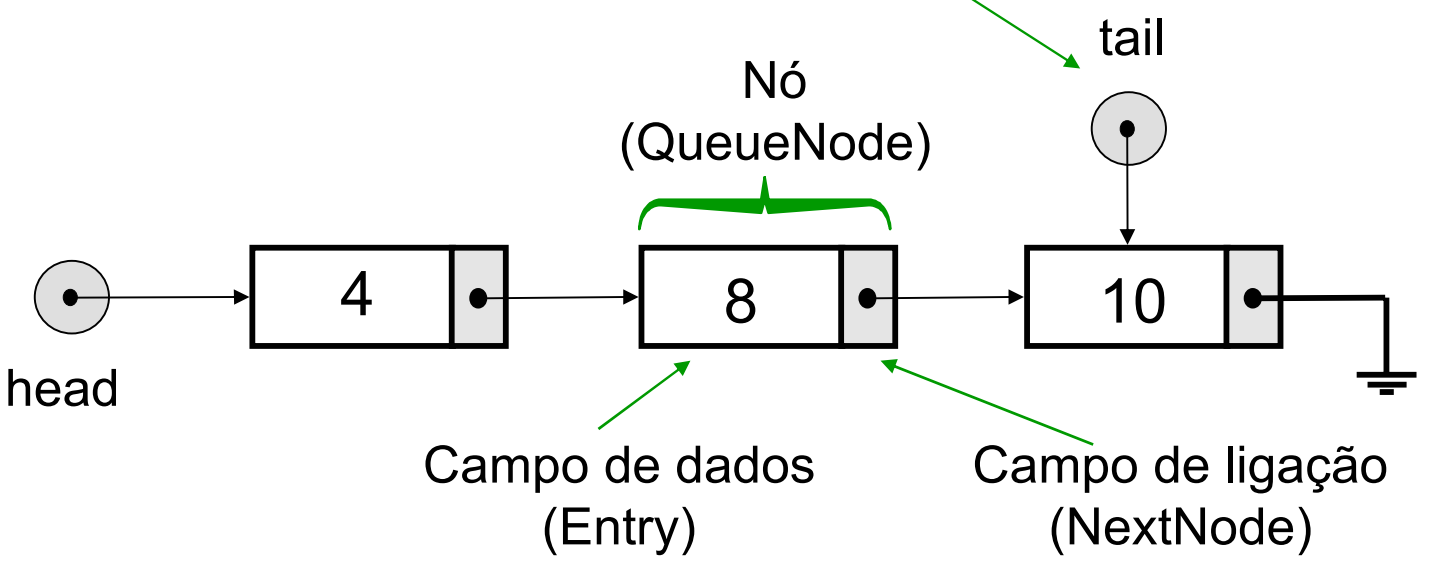


Detalhes de Implementação

- Nós precisamos armazenar o início e final da fila

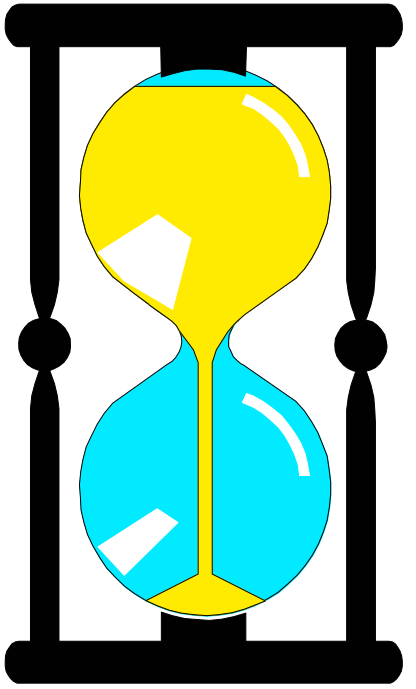


Outro ponteiro armazena o final da fila



Questão

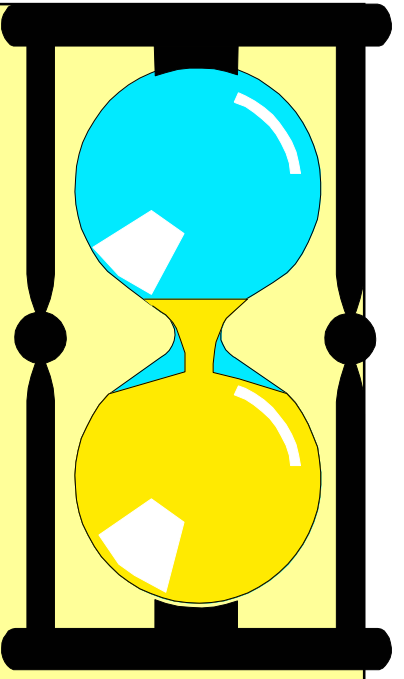
Utilize estas idéias para escrever uma declaração de tipo que poderia implementar uma fila encadeada. A declaração deve ser um objeto com dois campos de dados



Você tem 5 minutos
para escrever a declaração

Uma Solução

```
class Queue
{ public:
    Queue();
    ~Queue();
    void Append(int x);
    void Serve(int &x);
    bool Empty();
    bool Full();
  private:
    // declaração de tipos
    struct QueueNode
    { int Entry;                    // tipo de dado colocado na fila
      QueueNode *NextNode;          // ligação para próximo elemento na fila
    };
    typedef QueueNode *QueuePointer;

    // declaração de campos
    QueuePointer head, tail;
};
```



Uma Solução

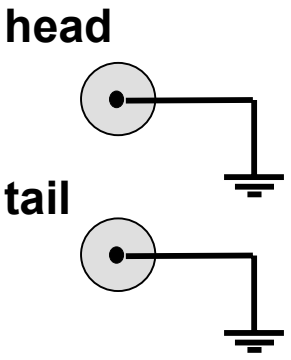
```
class Queue
{ public:
    Queue();
    ~Queue();
    void Append(int x);
    void Serve(int &x);
    bool Empty();
    bool Full();
  private:
    // declaração de tipos
    struct QueueNode
    { int Entry;                    // tipo de dado colocado na fila
      QueueNode *NextNode;          // ligação para próximo elemento na fila
    };
    typedef QueueNode *QueuePointer;

    // declaração de campos
    QueuePointer head, tail;
};
```

Observe que o tipo **QueueEntry** nesse caso é um inteiro

Construtor

Queue::Queue()

A Fila deve iniciar vazia...

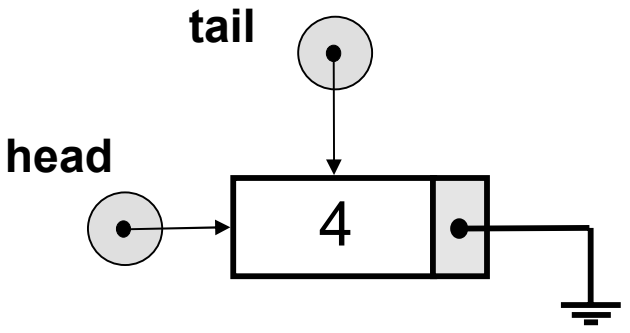



```
Queue::Queue()
{
    head = tail = NULL;
}
```

Destruidor

Queue::~Queue()

Usando alocação dinâmica, o destruidor deve retirar todos os elementos da fila enquanto ela não estiver vazia. Lembre-se que atribuir NULL a head ou tail não libera o espaço alocado anteriormente!

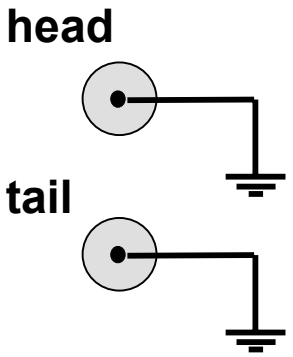



```
Queue::~Queue()
{ int x;
  while ( ! Empty() )
    Serve(x);
}
```

Status: Empty

bool Queue::Empty()

Lembre-se que a fila inicia
vazia, com head = tail = NULL...

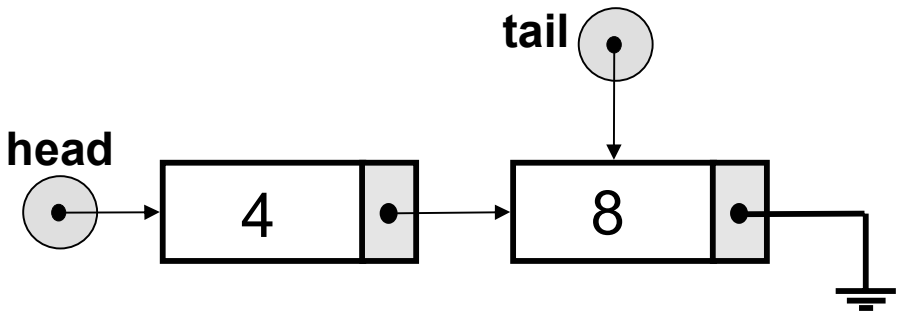



```
bool Queue::Empty()
{
    return (head == NULL);
}
```

Status: Full

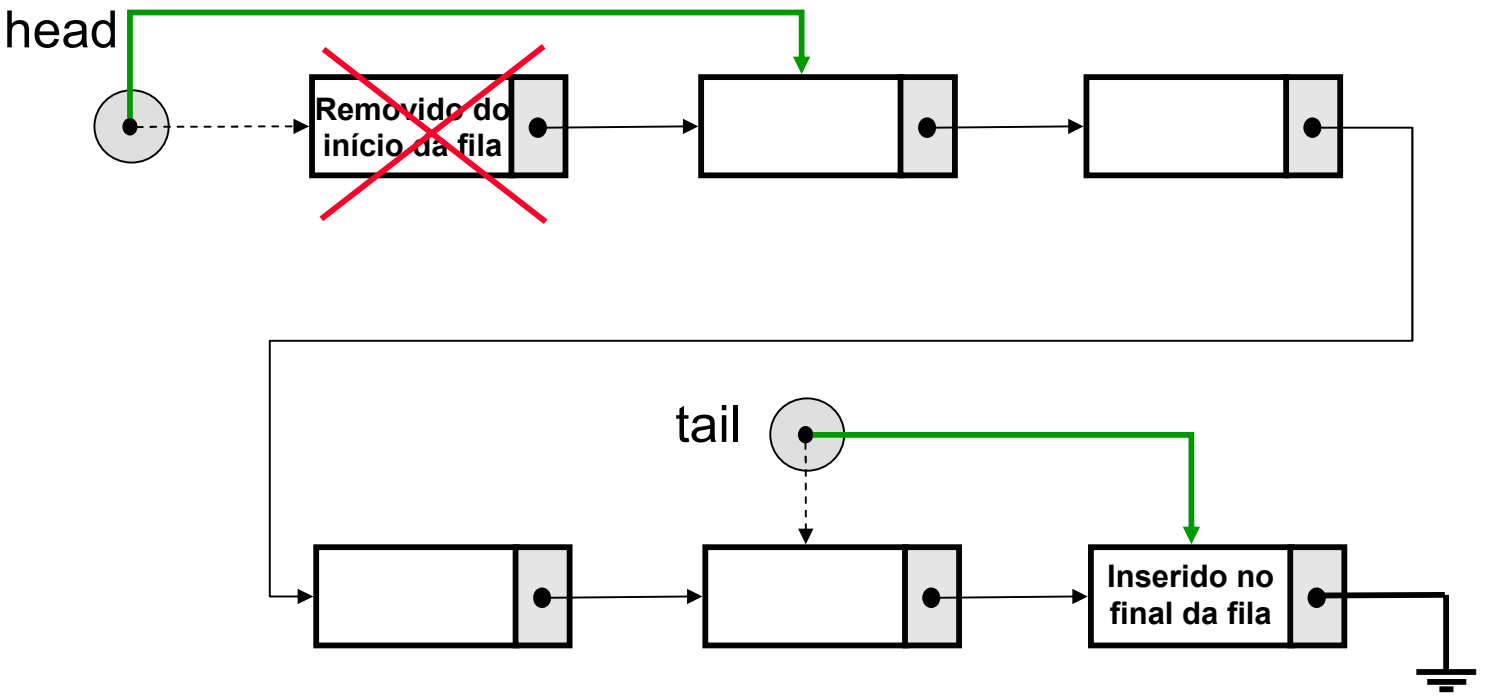
bool Queue::Full()

...e que não há limite quanto ao número máximo de elementos da fila




```
bool Queue::Full()
{
   return false;
}
```

Operações Básicas

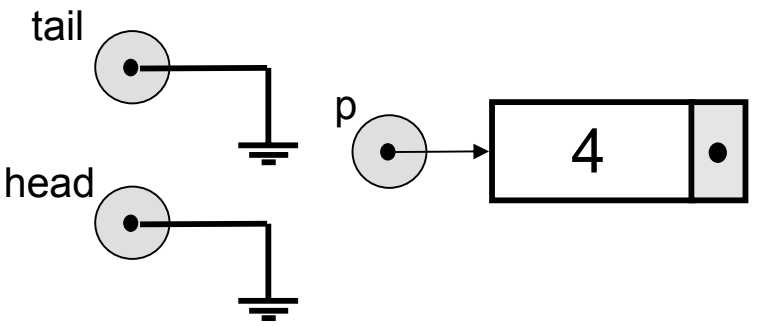


Operações Básicas: Append

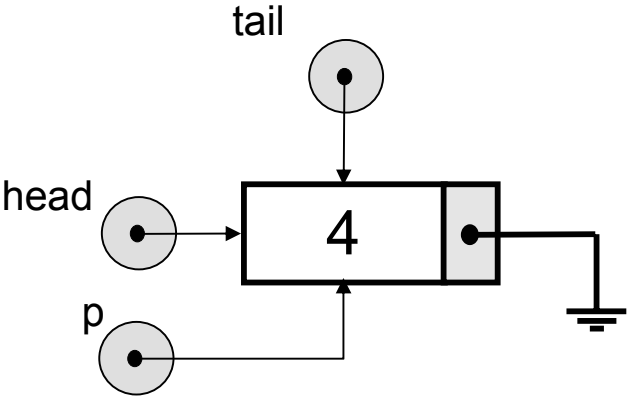
```
void Queue::Append(int x)
```

Considere agora uma fila vazia, o que significa **head** = **tail** = NULL e adicione o primeiro nó. Assuma que esse nó já foi criado em algum lugar na memória e pode ser localizado usando um ponteiro **p** do tipo QueuePointer

Fila Vazia



Fila de tamanho 1



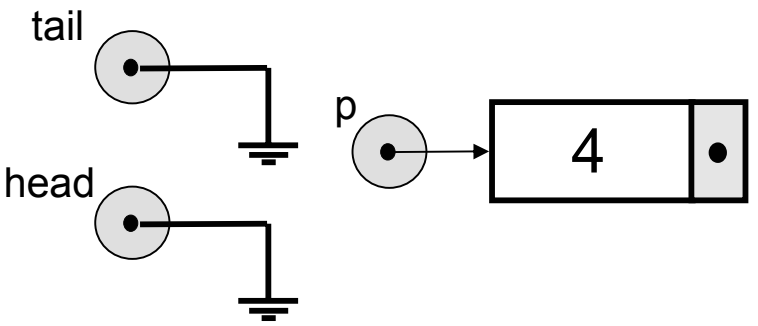
Operações Básicas: Append

```
void Queue::Append(int x)
```

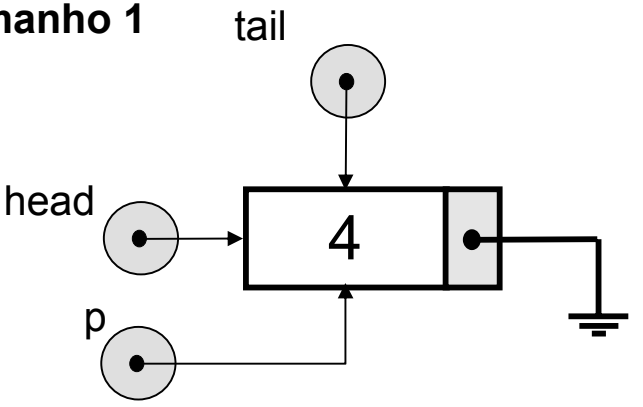
Assim, colocar (Append) o primeiro nó **p** na fila consiste nas instruções:

```
head = tail = p;
p->NextNode = NULL;
```

Fila Vazia



Fila de tamanho 1

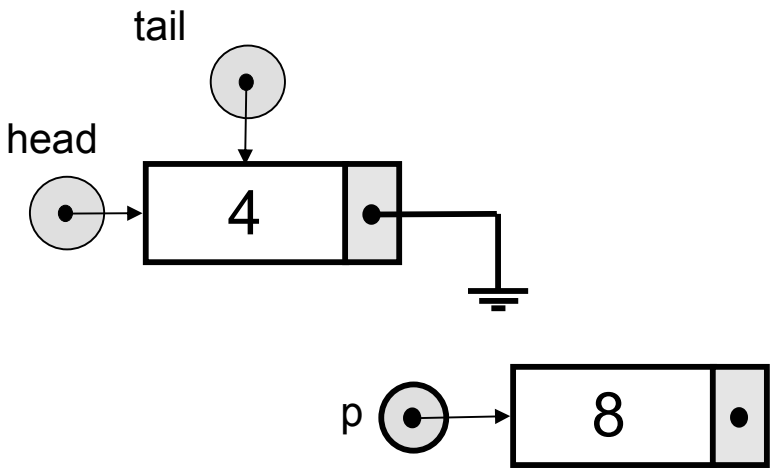


Operações Básicas: Append

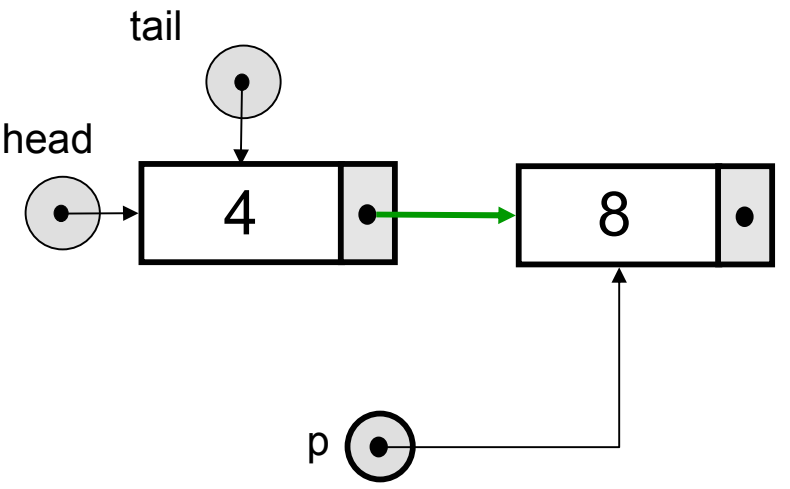
```
void Queue::Append(int x)
```

Assuma que o próximo nó a ser colocado na fila já foi criado em algum lugar na memória e pode ser localizado usando um ponteiro **p** do tipo QueuePointer. Alteramos o campo de ligação do último nó para **p**...

Fila de tamanho 1



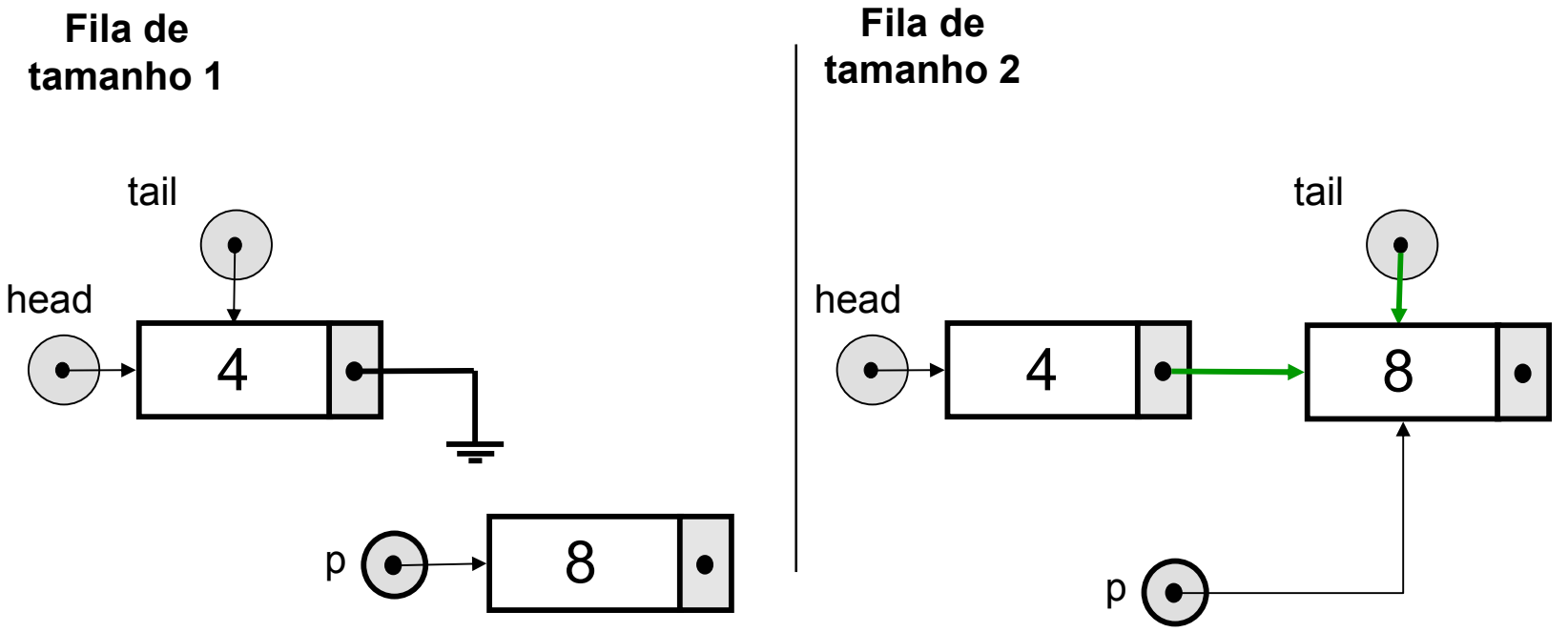
Fila de tamanho 2



Operações Básicas: Append

```
void Queue::Append(int x)
```

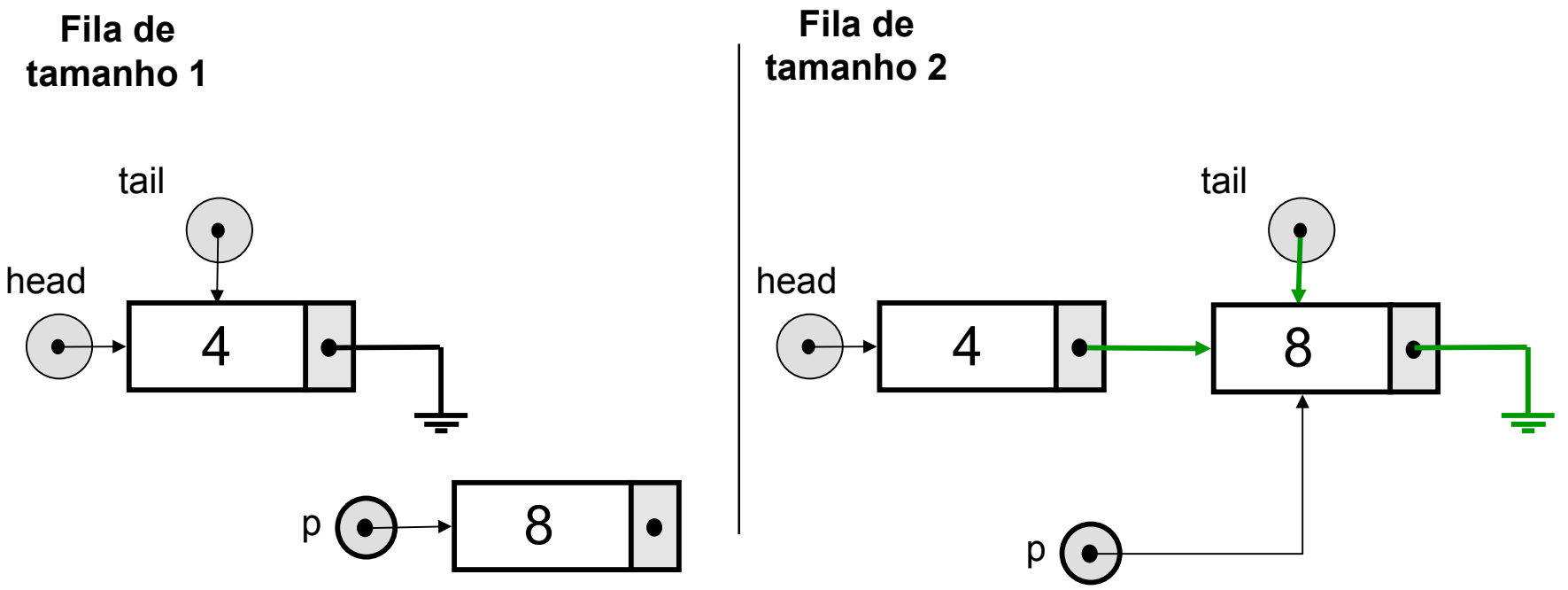
Alteramos o campo de ligação do último nó para **p**; em seguida mudamos **tail** para apontar para **p**...



Operações Básicas: Append

```
void Queue::Append(int x)
```

Alteramos o campo de ligação do último nó para **p**; em seguida mudamos **tail** para apontar para **p** e, finalmente, aterramos o último nó da lista



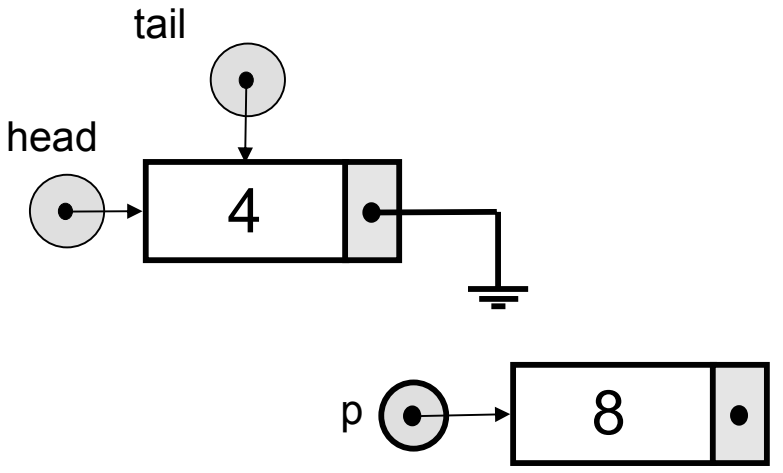
Operações Básicas: Append

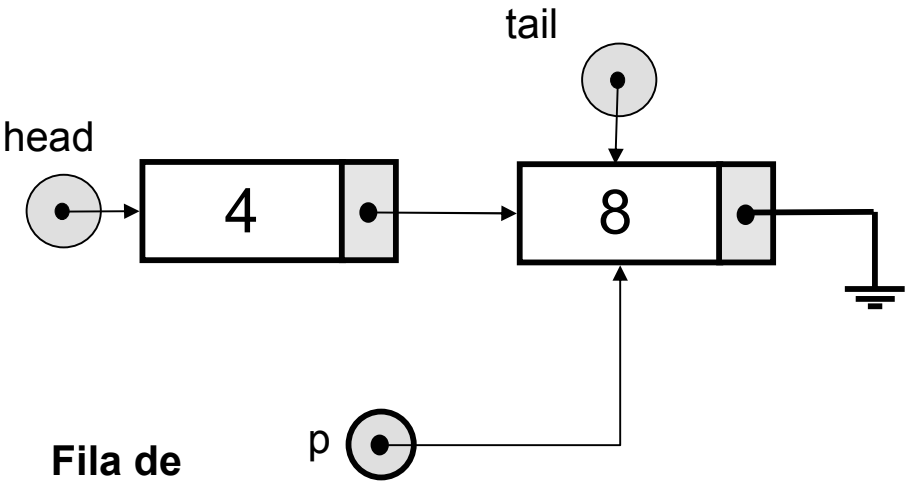
```
void Queue::Append(int x)
```

Resumindo, para colocar (Append) os demais nós (que não o primeiro) **p** na fila consiste nas instruções:

```
tail->NextNode = p;
tail = p;
p->NextNode = NULL;
```

Fila de tamanho 1



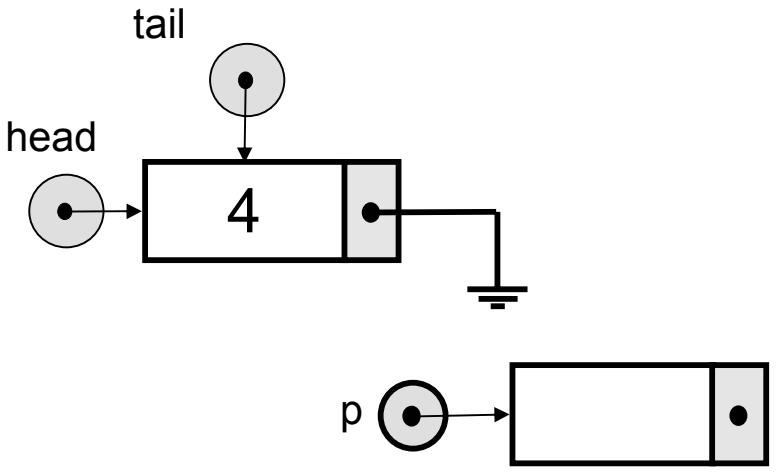


Fila de tamanho 2

Operações Básicas: Append

void Queue::Append(int x)

Inicialmente, alocamos o novo nó, usando o ponteiro **p**...

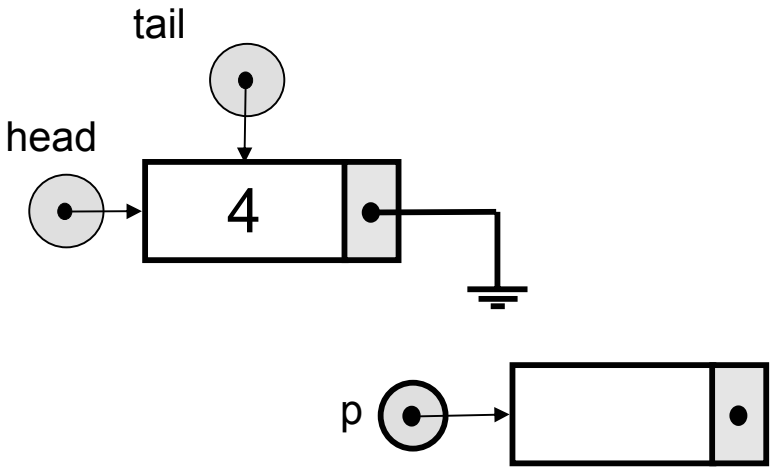



```
void Queue::Append(int x)
{ QueuePointer p;

  p = new QueueNode;
  ...
}
```

Operações Básicas: Append

void Queue::Append(int x)

Inicialmente, alocamos o novo nó, usando o ponteiro **p**. Se não houver espaço na memória, escrevemos uma mensagem de erro e terminamos

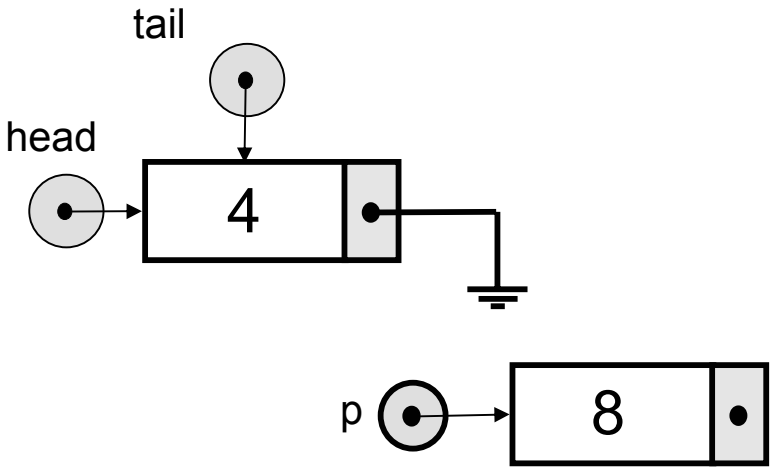



```
void Queue::Append(int x)
{ QueuePointer p;

  p = new QueueNode;
  if(p == NULL)
  { cout << “Memoria insuficiente”;
    abort();
  }
  ...
}
```

Operações Básicas: Append

void Queue::Append(int x)

Caso contrário, colocamos o elemento **x** no campo de dados de **p**

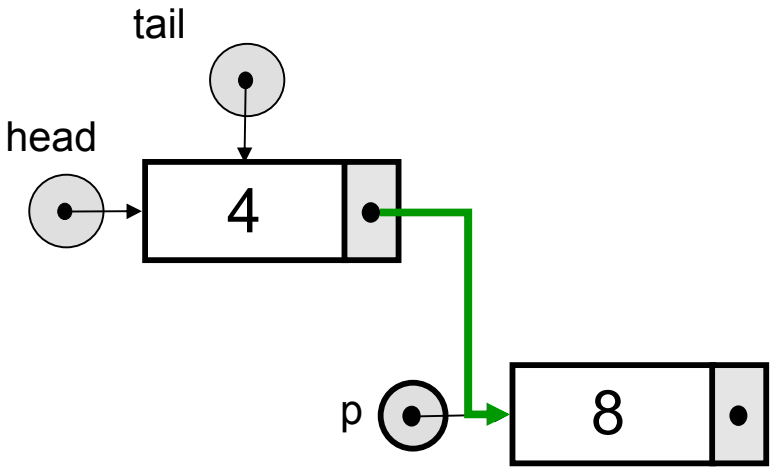



```
void Queue::Append(int x)
{ QueuePointer p;

  p = new QueueNode;
  if(p == NULL)
  { cout << “Memoria insuficiente”;
    abort();
  }
  p->Entry = x;
  ...
}
```

Operações Básicas: Append

void Queue::Append(int x)

Caso contrário, colocamos o elemento **x** no campo de dados de **p** e alteramos os ponteiros apropriadamente…

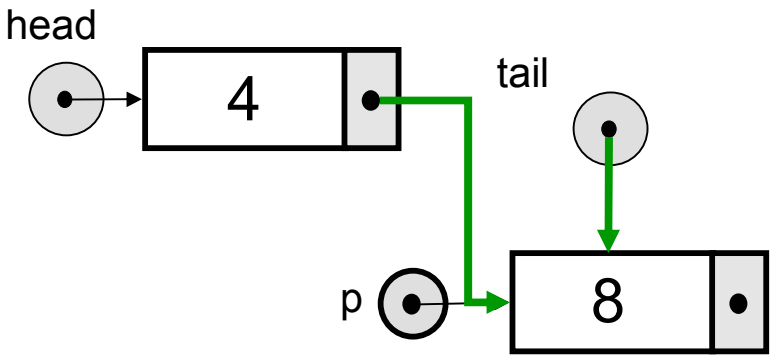



```
void Queue::Append(int x)
{ QueuePointer p;

  p = new QueueNode;
  if(p == NULL)
  { cout << “Memoria insuficiente”;
    abort();
  }
  p->Entry = x;
  if (Empty())
    head = tail = p;
  else
  { tail->NextNode = p;
    tail = p;
  }
  ...
}
```

Operações Básicas: Append

void Queue::Append(int x)

Caso contrário, colocamos o elemento **x** no campo de dados de **p** e alteramos os ponteiros apropriadamente…

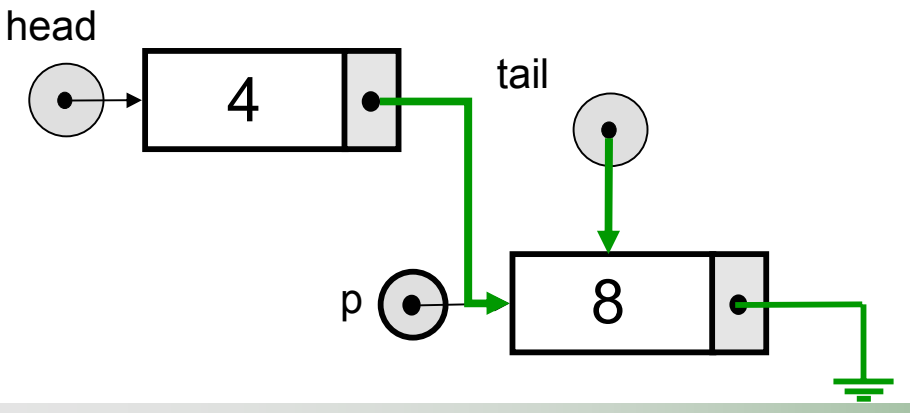



```
void Queue::Append(int x)
{ QueuePointer p;

  p = new QueueNode;
  if(p == NULL)
  { cout << “Memoria insuficiente”;
    abort();
  }
  p->Entry = x;
  if (Empty())
    head = tail = p;
  else
  { tail->NextNode = p;
    tail = p;
  }
  ...
}
```

Operações Básicas: Append

void Queue::Append(int x)

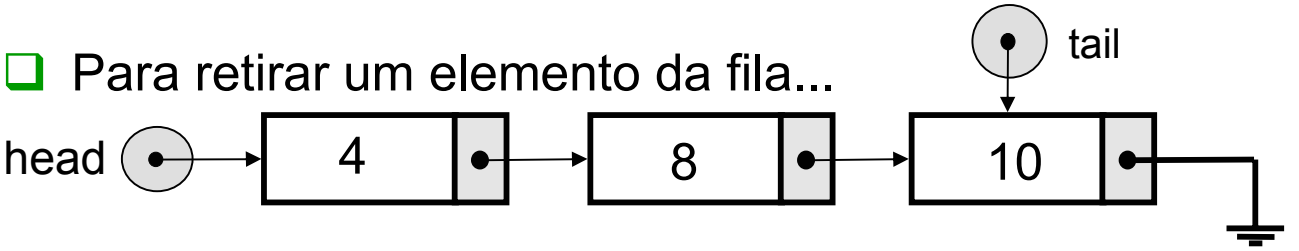
Caso contrário, colocamos o elemento **x** no campo de dados de **p** e alteramos os ponteiros apropriadamente. Finalmente, aterramos o último nó da fila




```
void Queue::Append(int x)
{ QueuePointer p;

  p = new QueueNode;
  if(p == NULL)
  { cout << “Memoria insuficiente”;
    abort();
  }
  p->Entry = x;
  if (Empty())
    head = tail = p;
  else
  { tail->NextNode = p;
    tail = p;
  }
  p->NextNode = NULL;
}
```

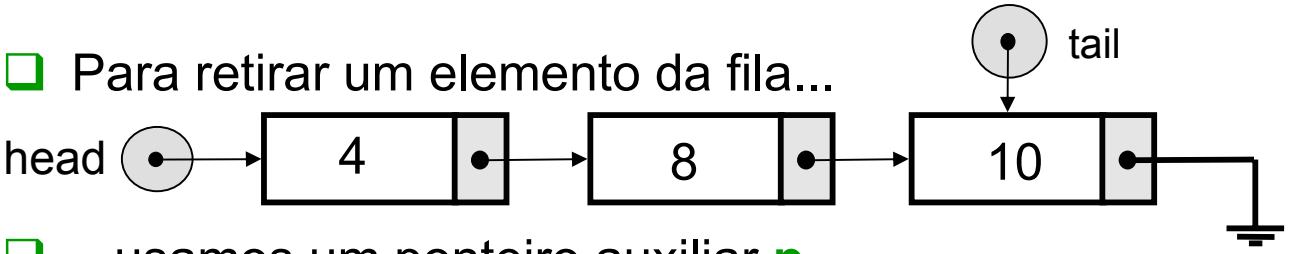
Operações Básicas: Serve

- Para retirar um elemento da fila...

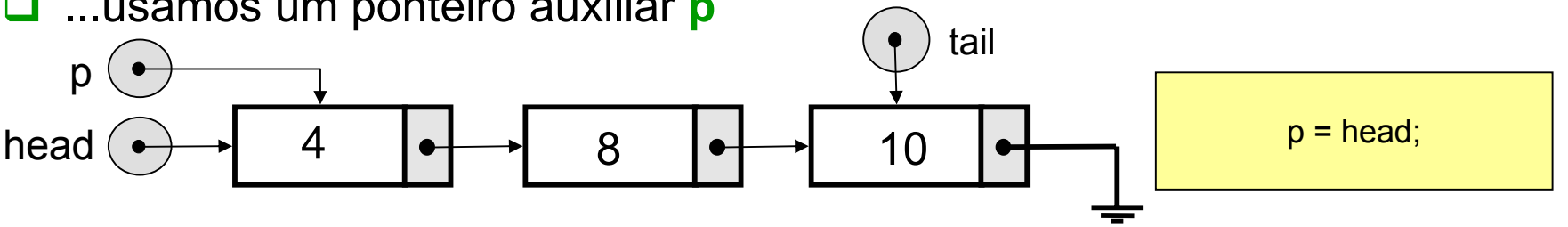


Operações Básicas: Serve

- Para retirar um elemento da fila...

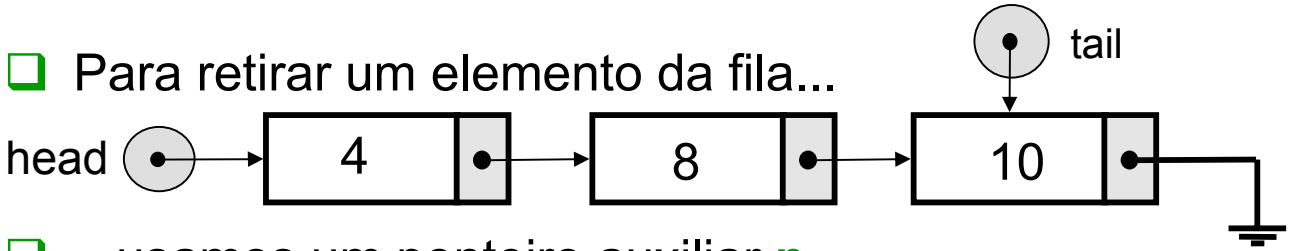


- ...usamos um ponteiro auxiliar **p**

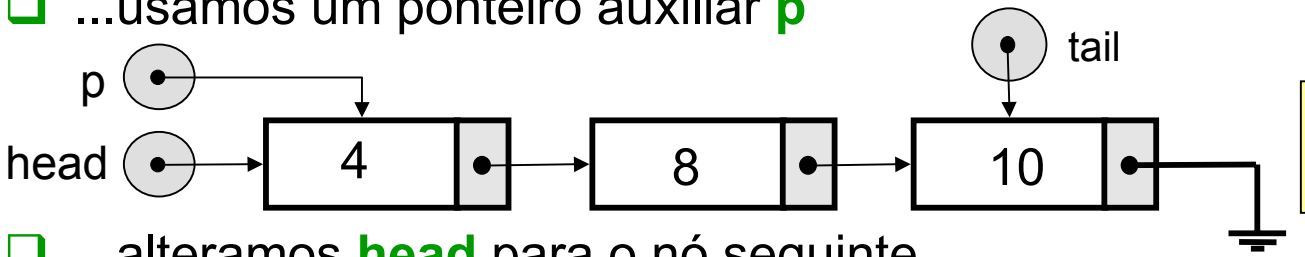


Operações Básicas: Serve

- Para retirar um elemento da fila...

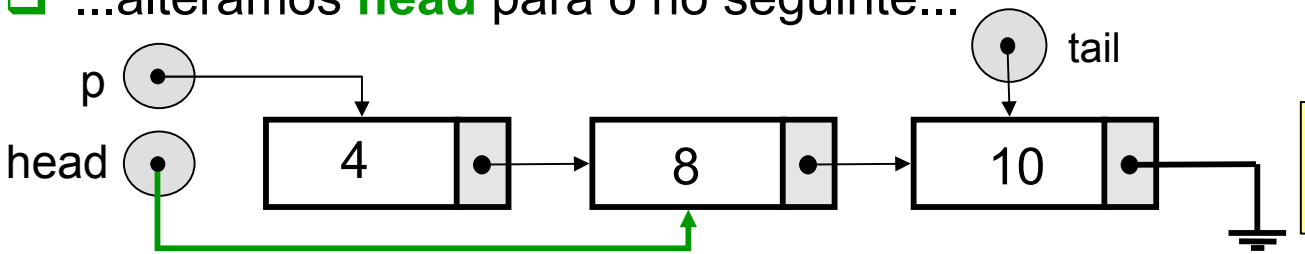


- ...usamos um ponteiro auxiliar **p**



p = head;

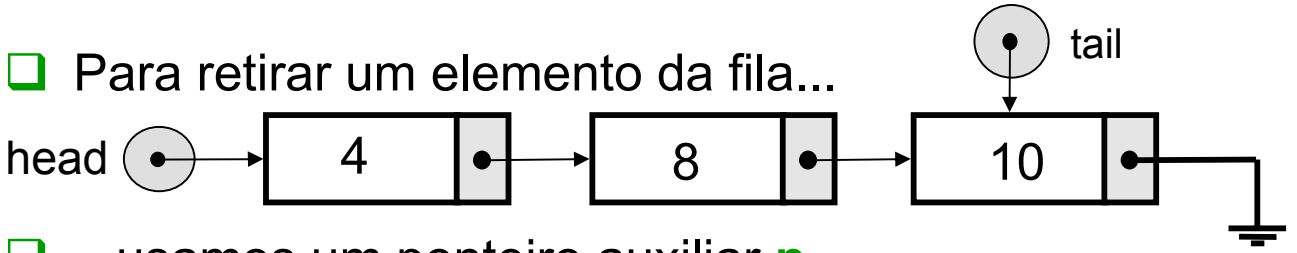
- ...alteramos **head** para o nó seguinte...



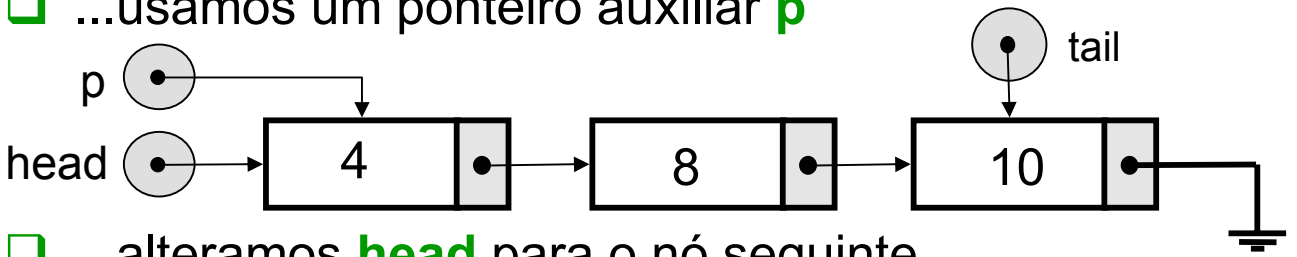
head = head->NextNode

Operações Básicas: Serve

- Para retirar um elemento da fila...

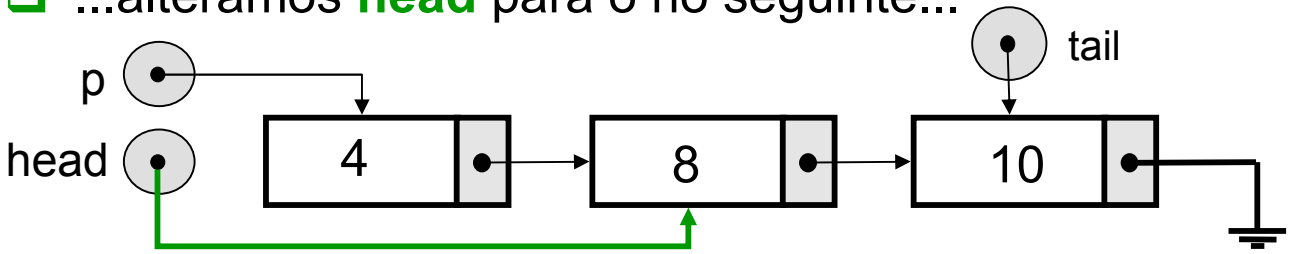


- ...usamos um ponteiro auxiliar **p**



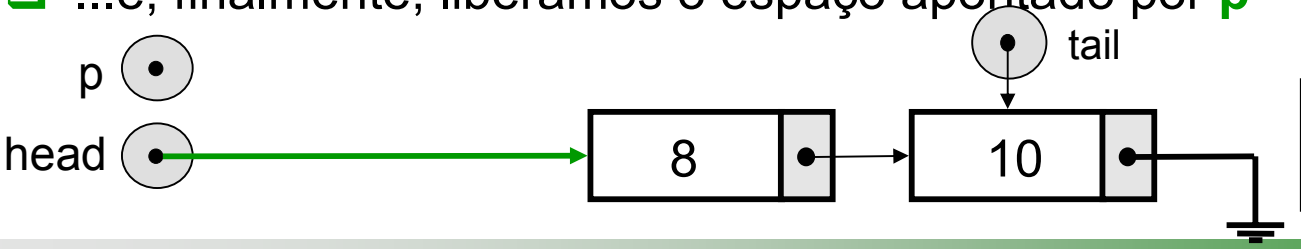
p = head;

- ...alteramos **head** para o nó seguinte...



head = head->NextNode

- ...e, finalmente, liberamos o espaço apontado por **p**

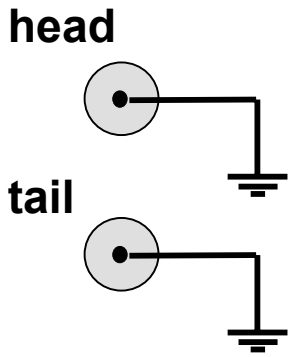


delete p;

Operações Básicas: Serve

void Queue::Serve(int &x)

Inicialmente, verificamos se a fila está vazia. Em caso afirmativo, imprimimos uma mensagem de erro e terminamos

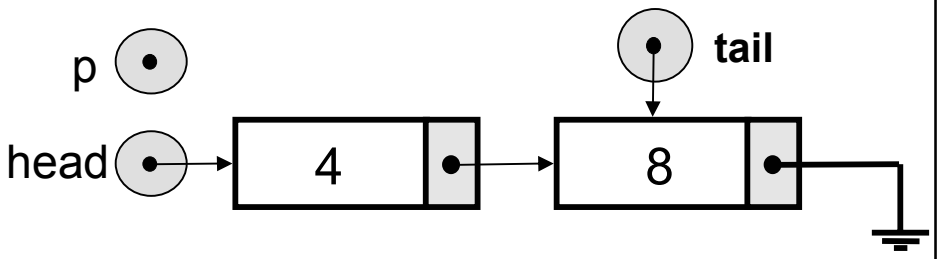


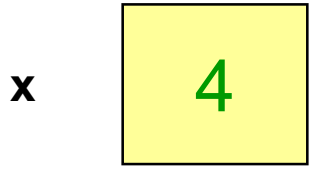

```
void Queue::Serve(int &x)
{ QueuePointer p;

  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
  ...



}
```

Operações Básicas: Serve

void Queue::Serve(int &x)

Caso contrário, armazenamos o valor do início da fila na variável **x**



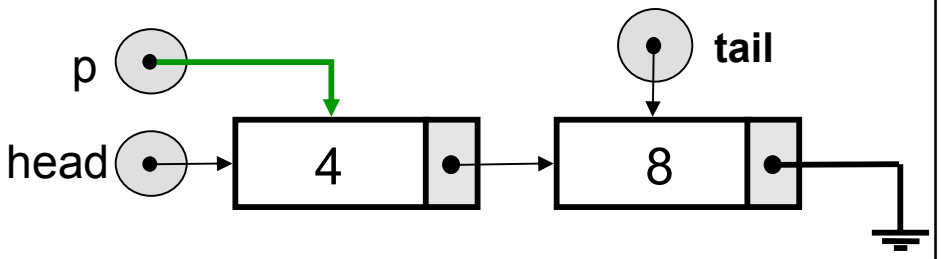


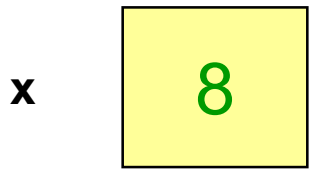

```
void Queue::Serve(int &x)
{ QueuePointer p;

  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
   x = head->Entry;
   ...


}
```

Operações Básicas: Serve

void Queue::Serve(int &x)

Apontamos o ponteiro auxiliar **p** para o início da fila...



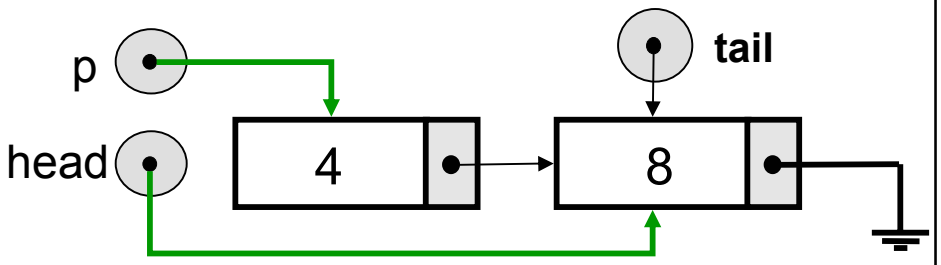


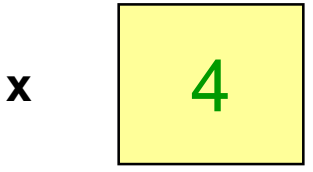

```
void Queue::Serve(int &x)
{ QueuePointer p;

  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
   x = head->Entry;
   p = head;
   ...

}
```

Operações Básicas: Serve

void Queue::Serve(int &x)

Apontamos o ponteiro auxiliar **p** para o início da fila; alteramos **head** para o próximo nó...



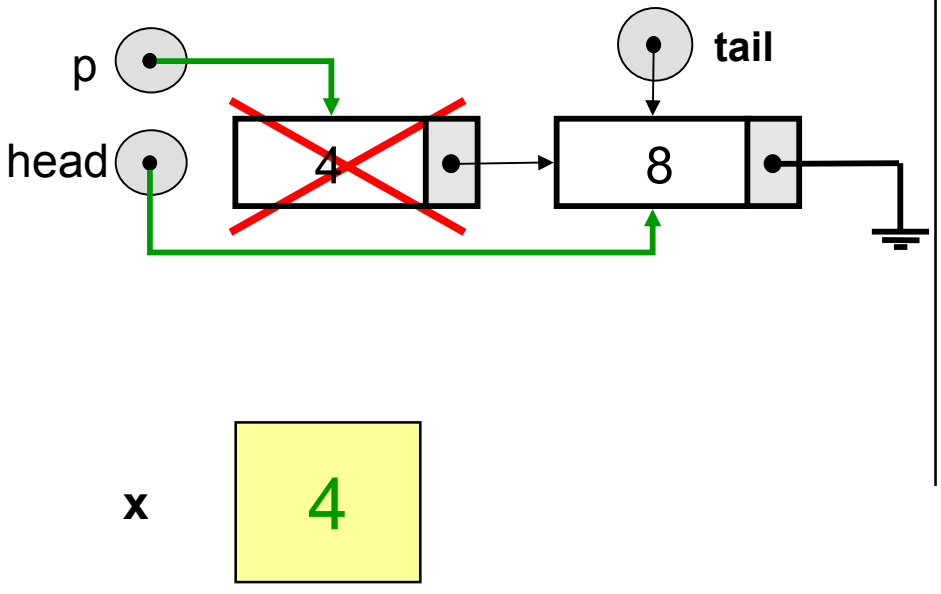



```
void Queue::Serve(int &x)
{ QueuePointer p;

  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
   x = head->Entry;
   p = head;
   head = head->NextNode;
   ...
}
```

Operações Básicas: Serve

void Queue::Serve(int &x)

Finalmente, liberamos o espaço apontado por **p**

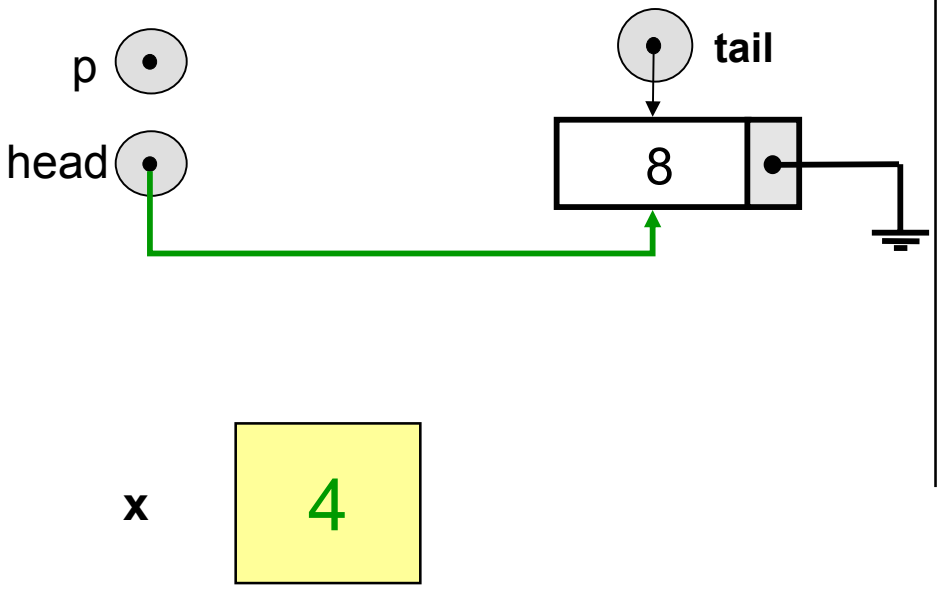



```
void Queue::Serve(int &x)
{ QueuePointer p;

  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
   x = head->Entry;
   p = head;
   head = head->NextNode;
   delete p;
   ...
}
```

Operações Básicas: Serve

void Queue::Serve(int &x)

Finalmente, liberamos o espaço apontado por **p** cuidando que, caso a fila fique vazia o ponteiro **tail** reflita também esse fato




```
void Queue::Serve(int &x)
{ QueuePointer p;

  if (Empty())
   {  cout << “Fila Vazia”;
      abort();
   }
   x = head->Entry;
   p = head;
   head = head->NextNode;
   delete p;
   if (head == NULL)
     tail = NULL;
}
```

Exercícios

- Implemente Clear(), usando apenas Serve() e Empty()
- Implemente Clear() trabalhando diretamente com ponteiros
- Implemente o destruidor da fila usando diretamente ponteiros
- Implemente Front() e Rear()
- Implemente Size()
- É possível diminuir o tempo necessário para calcular o tamanho da fila utilizando Size()? O que é necessário para isso?

Solução Clear

- Usando apenas Serve e Empty

```
void Queue::Clear()
{ int x;

  while(! Empty())
    Serve(x);
}
```

- Utilizando campos do objeto

```
void Queue::Clear()
{ QueuePointer p;

   while(head != NULL)
   { p = head;
     head = head->NextNode;
     delete p;
   }
   tail = NULL;
}
```

Solução Front/Read

```
void Queue::Front(int &x)
{  if(Empty())
   { cout << “Fila vazia”;
     abort();
   }
   x = head->Entry;
}
```

```
void Queue::Rear(int &x)
{  if(Empty())
   { cout << “Fila vazia”;
     abort();
   }
   x = tail->Entry;
}
```

Solução Size

```
int Queue::Size()
{  int count=0;
   QueuePointer p;

   p = head;
   while (p != NULL)
   { count++;
     p = p->NextNode;
   }
   return count;
}
```