CENTRO UNIVERSITÁRIO FEI

Ciência da computação

RELATÓRIO CE4652: Estrutura de Dados

São Bernardo do Campo

2020

ANDY SILVA BARBOSA, RA: 22.218.025-9

VITOR ACOSTA DA ROSA, RA: 22.218.006-9

RELATÓRIO CE4652: Estrutura de Dados

Professor: Guilherme Alberto Wachs Lopes

Disciplina: CE4652 – Estrutura de Dados

São Bernardo do Campo

2020

1. FILA ESTÁTICA CIRCULAR

1.1. Descrição

A fila estática circular é um vetor de tamanho definido, estático. Suas operações seguem o princípio FIFO (First In First Out), ou seja, obedece a ordem de chegada e o primeiro valor a ser inserido na fila, será também o primeiro a sair. É chamada de circular pois o último valor da fila terá como próximo valor o primeiro, e o primeiro terá como valor anterior o último, gerando um “círculo”.

1.2. Lógica da fila

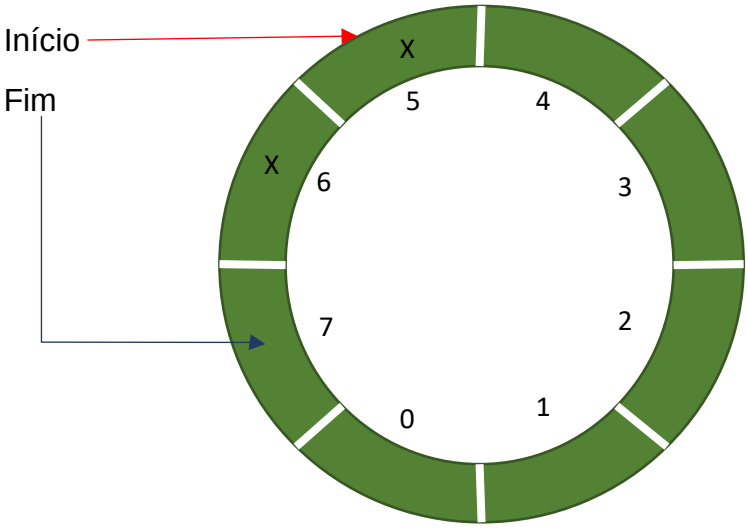
Para que a fila funcione de maneira circular, existem duas variáveis, representadas aqui como $InicioFila$ e $FimFila$, que funcionam como “ponteiros” e que armazenam os índices de início e fim da lista, respectivamente. Com isso, a movimentação dos valores do vetor é feita a partir das funções: enfileira e desenfileira; ambas, movem os “ponteiros” pela fila, de acordo com a necessidade.

Em toda função que altere os elementos da fila, é necessário mover os “ponteiros” de início e fim. Essa movimentação é realizada através de um cálculo da seguinte forma: a operação *mod* (módulo) é realizada a partir dos “ponteiros” de início **ou** fim (dependendo da função) com o tamanho máximo da fila, como mostra a fórmula abaixo:

$$fim/inicioFila \leftarrow (fim/inicioFila + 1) \ \mathbf{mod} \ TamanhoMáximo$$

Fórmula de circularidade

Essa fórmula é aplicada pois o incremento das variáveis que representam os índices de começo e/ou fim deve ser circular, ou seja, quando é requisitado o incremento e uma das variáveis encontra-se no fim da fila, essa variável deslocará para o começo, mantendo o ciclo. A figura 1 exemplifica a utilização dessa fórmula.



Nesse caso, temos uma fila com o $TamanhoMáximo = 8$. Note que, o início da fila é dado no índice 5, e o fim no índice 7. Caso o usuário desejasse inserir um novo elemento, se somente incrementássemos a variável “fim”, teríamos o valor 8, que é um índice inválido.

Aqui, é possível verificar como a fórmula citada resolve esse problema.

Com a variável “fim” contendo o valor 7, temos o seguinte:

$$fimFila \leftarrow (7 + 1) \textbf{ mod } 8$$

$$fimFila \leftarrow 0$$

O que faz sentido, pois agora, o fim da fila “aponta” para o índice zero, e manteve o ciclo da estrutura. Sendo possível inserir novos valores.

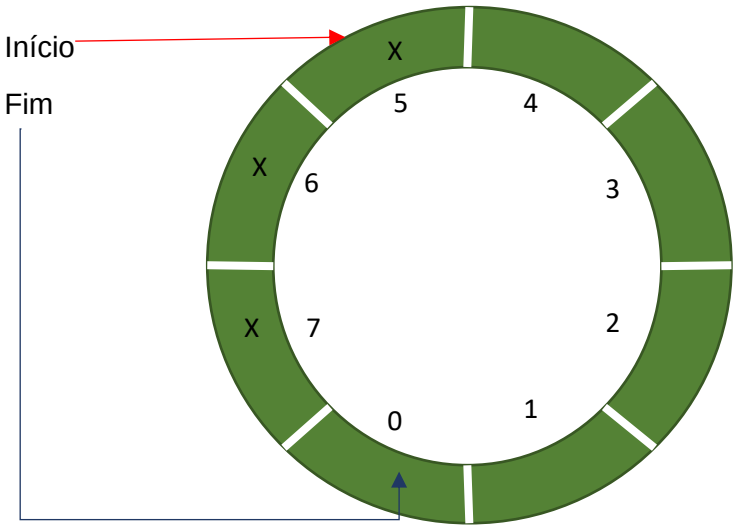


Figura 1 – Circularidade de um fila estática circular

Note que o tamanho máximo é referente ao tamanho da fila, fixo, definido em código. Note também que, independentemente da variável que será considerada ($InicioFila$ ou $FimFila$), sempre será somado 1, isso porque, essa estrutura deve conter um espaço vazio entre o fim e começo da fila, esse espaço é chamado de sentinela e tem como função verificar se a fila ainda pode conter novos elementos e impede que o $InicioFila$ e $FimFila$ sejam iguais (caso que só ocorre com a fila vazia).

1.3. Método “enfileirar”

A função para inserir dados na fila, o método “enfileira”, tem como objetivo alocar o valor digitado pelo usuário em uma fila estática circular. Sabendo que o tamanho do vetor é estático, e que a inserção será feita sempre no último index possível, o processo de inserção tem o custo operacional baixo, $O(1)$.

Antes do número entrar na fila, é verificado se o sentinela, dado por $FimFila + 1$, está na mesma posição do início da fila, ou seja, ($FimFila + 1 = InicioFila$), caso esteja, não é possível alocar o novo valor na fila, pois o vetor está cheio.

Caso contrário, o novo número será colocado na última posição do vetor, representada pelo $FimFila$, além disso, tal “ponteiro” receberá o valor da fórmula de circularidade, apontando para próxima posição. O Pseudocódigo da inserção é mostrado a seguir:

```
função enfileira: logico
    var numero: inteiro
    inicio
    se (fimFila + 1) MOD (tamanho) = inicioFila então
       enfileira ← FALSO

    fila[fimFila] ← numero
    fimFila ← (fimFila+1) MOD (tamanho)
    enfileira ← VERDADEIRO
    fimEnfileira
```

1.4. Método “desenfileira”

A função que “remove” valores da fila, o método “desinfileira”, tem como objetivo desalocar o primeiro valor inserido pelo usuário na fila estática circular, após utilizar esse método, o primeiro valor não será excluído, mas sim, desconsiderado da fila. Sabendo que o tamanho do vetor é estático, e que sempre o valor a ser excluído está no começo do vetor, o processo de remoção tem o custo operacional baixo, $O(1)$.

Antes de retirar o primeiro valor inserido, é verificado se os “ponteiros” estão na mesma posição ($FimFila = InícioFila$), caso esteja, não é possível desalocar o primeiro valor, pois o vetor está vazio.

Caso contrário, é criada uma variável de controle que possui o valor sendo o primeiro valor inserido pelo usuário na fila, após isso, o $InícioFila$ deslocará para o próximo valor da lista, seguindo a fórmula de circularidade, fazendo com que o valor desalocado não apareça mais na fila. O Pseudocódigo de desenfileirar é representado abaixo:

```
função desinfileira: logico,inteiro
    var temp: inteiro
    inicio
    se inicioFila = fimFila então
       desinfileira ← FALSO,-1

    temp ← fila[inicioFila]
    inicioFila ← (inicioFila+1) MOD (tamanho)
    desinfileira ← VERDADEIRO, temp
    fimDesinfileira
```

1.5. Método de busca

A função Busca tem como objetivo verificar se o valor inserido pelo usuário pertence ao vetor. A verificação é feita percorrendo todo o vetor, de ponta a ponta, e comparando se cada valor é igual ao inserido pelo usuário. Como a busca é uma busca linear, seu custo operacional é dado por $O(n)$. O pseudocódigo desse método é representado abaixo:

```
função busca: inteiro
        var j: inteiro
        inicio
        para cada j ← 0, j < tamanho, j ← j+1 então
           se fila[j] = valor então
                busca ← j
        busca ← -1

        fimBusca
```

1.6. Método de impressão

A função imprime tem como objetivo apresentar ao usuário todas as inserções feitas por ele na ordem de inserção. Para tal, a impressão de cada valor do vetor começa onde a variável $InícioFila$ está “apontando” indo até o índice “apontado” pelo $FimFila$. Vale ressaltar que essa operação realiza para cada nova iteração o cálculo com a fórmula de circularidade citada anteriormente. Como o vetor é percorrido item a item até o final, seu custo operacional é dado por $O(n)$. O pseudocódigo pode ser verificado abaixo:

```
função imprime: escreve
        var i: inteiro
        inicio
        para cada i ←inicioFila, i ≠ fimFila, i ← (i + 1) MOD (tamanho) então
           escreve(fila[i])
        escreve('\n')

        fimImprime
```

1.7. Pseudocódigo de uma Fila Estática Circular

```
Algoritmo "Fila Estática Circular"

classe Fila
   inicio classe
```

```
função construtor
    inicio
    var fila : vetor
    var inicioFila, fimFila, tamanho: inteiro

    inicioFila ← 0
    fimFila ← 0
    tamanho ← 8

    fimConstrutor

função enfileira: logico
    var numero: inteiro
    inicio
    se (fimFila + 1) MOD (tamanho) = inicioFila então
       enfileira ← FALSO

    fila[fimFila] ← numero
    fimFila ← (fimFila+1) MOD (tamanho)
    enfileira ← VERDADEIRO
    fimEnfileira

função desinfileira: logico,inteiro
    var temp: inteiro
    inicio
    se inicioFila = fimFila então
       desinfileira ← FALSO,-1

    temp ← fila[inicioFila]
    inicioFila ← (inicioFila+1) MOD (tamanho)
    desinfileira ← VERDADEIRO, temp
    fimDesinfileira
```

```
função busca: inteiro
        var j: inteiro
        inicio
        para cada j ← 0, j < tamanho, j ← j+1 então
           se fila[j] = valor então
                busca ← j
        busca ← -1

        fimBusca

função imprime: escreve
       var i: inteiro
       inicio
       para cada i ←inicioFila, i ≠ fimFila, i ← (i + 1) MOD (tamanho) então
          escreve(fila[i])
       escreve('\n')

      fimImprime
   fimClasse
```

1.8. Código em JavaScript de uma Fila Estática Circular

```javascript
class Fila{
    constructor(){
        this.fila = [];
        this.begin = 0;
        this.end = 0;
        this.tamanho = 8;
    }

    enfileira(numero){
        if((this.end+1)%(this.tamanho) === this.begin){
            return false;
        }

        this.fila[this.end] = numero;
        this.imprime();

        this.end = (this.end+1) % (this.tamanho);

        return true;
    }

    desinfileira(){
        if(this.begin === this.end){
            return [false, -1];
        }
        let temp = this.fila[this.begin];
        this.fila[this.begin] = " ";
        this.begin = (this.begin + 1) % (this.tamanho);
        return [true,temp];
    }

    busca(valor){
        for(var j = 0; j < this.tamanho; j++){
          if(this.fila[j] === valor){
            return j;
          }
        }
        return -1;
    }

    limpa(){
        for(let i=0; i < this.tamanho; i++){
            this.fila[i] = " ";
        }
        this.begin = this.end;
    }

    imprime(){
        for(var i = this.begin; i != this.end; i=(i + 1) % (this.tamanho)){
            console.log(this.fila[i]);
        }
        console.log('Fim Impressão');
    }
}
```

Figura 2 – Código fonte na linguagem JavaScript

2. LISTA DINÂMICA DUPLAMENTE ENCADEADA (LDDE)

2.1. Descrição

Trata-se de uma estrutura linear dinâmica de números reais ou inteiros, ordenados de forma crescente, que aumenta sua capacidade de acordo com a necessidade do usuário. Por ser uma lista duplamente encadeada, cada elemento pertencente a ela possui uma referência do próximo elemento e também do anterior.

2.2. Lógica da lista

A estrutura baseia-se em nós, onde cada nó de uma LDDE, além da informação principal, guarda também dois ponteiros, para o próximo nó da lista e para o nó anterior. Nessa perspectiva, cada elemento pode tanto encontrar elementos anteriores como posteriores. Vale ressaltar que os elementos não estão alocados de forma sequencial na memória, e que existe uma variável de controle, representada aqui como n para armazenar o tamanho (ou quantidade de nós) existente nessa lista.

Como cada nó é alocado na memória, a única forma de percorrer essa lista é através dos ponteiros. Mais importantes que os ponteiros que indicam o próximo nó e o anterior de cada elemento, são os ponteiros que indicam o nó do começo da lista e o nó do fim da lista, esses, podem ser acessados imediatamente quando requerido. Através deles, é possível percorrer todos elementos, de ponta a ponta. A figura 3 demonstra como cada elemento está ligado nessa estrutura.

Figura 3 – Uma LDDE com três elementos

Vale ressaltar que, no exemplo acima, o X do nó de valor 12 e de valor 99, representa o ponteiro para *NULL*, através dessas indicações é possível verificar qual o começo da lista e qual o fim dela. As setas entre os nós representam os ponteiros, os quais possuem um caminho de ida (por exemplo

do nó 12 ao nó 57) e o caminho de volta (do nó 57 para o nó 12), possibilitando acessar tanto os nós posteriores como os anteriores a partir do atual.

2.3. Método de inserção

Todo novo nó a ser inserido, por padrão possui os apontamentos de próximo nó e nó anterior como nulos, como a lista não possui tamanho definido, a única condição que pode cancelar a inserção, é a falta de memória para alocar um novo nó.

Em toda nova inserção, são definidos ponteiros que representam a movimentação de valores na lista, representados aqui como $noAnterior$ e $noAtual$. O primeiro por padrão é definido como nulo, e tem como função armazenar a posição de memória na qual o último nó lido pelo algoritmo está, já o segundo, por padrão receberá o primeiro nó da lista, a fim de percorrê-la do começo ao final.

A inserção contém três cenários distintos: o novo nó pode ser inserido tanto no começo, como no meio ou final da lista. Toda inserção na estrutura, também redefine os ponteiros, do próprio elemento (ponteiros para próximo e anterior) e da classe (os ponteiros de início e fim da lista). A inserção, em relação ao custo computacional, pode ter no pior dos casos, $O(n)$. Vale ressaltar que, se o valor a ser inserido está no início da lista, o custo computacional é $O(1)$. O pseudocódigo para inserir novos nós na lista é representado abaixo:

```
função insere: logico
    var valor: inteiro
    inicio
    var novoNo: instancia de classe
    novoNo ← new No(valor)

    se novoNo = NULL então
       insere ← FALSO

    var noAnterior, noAtual

    noAnterior ← NULL
```

```
noAtual ← primeiro

enquanto noAtual ≠ NULL e ((noAtual.valor - valor) < 0) então
    noAnterior ← noAtual
    noAtual ← noAtual.proximo

se noAnterior ≠ NULL então
    noAnterior.proximo ← novoNo
    novoNo.anterior ← noAnterior
se não
    primeiro ← novoNo

se noAtual ≠ NULL então
   noAtual.anterior ← novoNo
se não
   ultimo ← novoNo

novoNo.proximo ← noAtual
n ← n+1
insere ← VERDADEIRO

fimInsere
```

2.4. Método de remoção

Para remover itens da LDDE, o processo envolve percorrer todos os nós à procura do nó que possui o valor igual ao que o usuário deseja deletar. Após percorrer os dados, caso chegue ao fim da lista sem encontrar o nó que possui o mesmo valor, a remoção não pode ocorrer, pois o elemento não existe. Caso o elemento seja encontrado na LDDE, o processo de refazer os ponteiros assemelha-se à inserção, contendo também os ponteiros de *noAnterior* e *noAtual*.

Além disso, existem também três casos possíveis: remoção no começo da lista, no meio e no fim. Como é necessário percorrer toda a estrutura, nó a nó, o custo computacional, pode ter no pior dos casos, $O(n)$. Vale ressaltar que,

se o valor a ser removido está no início da lista, o custo computacional é $O(1)$. O pseudocódigo de remoção pode ser visualizado abaixo:

```
função remove: logico
    var valor, i: inteiro
    var atual, anterior

    inicio
    atual ← primeiro
    anterior ← NULL

    para cada i ← 0, i < n e atual.valor ≠ valor, i ← i+1 então
        anterior ← atual
        atual ← atual.proximo

    se atual = NULL ou atual.valor ≠ valor então
        remove ← FALSO

    se anterior ≠ NULL então
        anterior.proximo ← atual.proximo
    se não
        primeiro ← atual.proximo

    se atual.proximo ≠ NULL então
        atual.proximo.anterior ← atual.anterior
    se não
        ultimo ← anterior

    n ← n-1
    remove ← VERDADEIRO

    fimRemove
```

2.5. Método de busca

Para buscar um valor na LDDE, a verificação é feita percorrendo toda a lista acessando os endereços de cada nó, de ponta a ponta, e comparando se cada valor do nó é igual ao inserido pelo usuário. Como a busca é uma busca linear, seu custo operacional é dado, no pior dos casos, por $O(n)$. O pseudocódigo desse método é representado abaixo:

```
função busca: inteiro
        var temp
        var valor, i: inteiro
        inicio
        temp ← primeiro
        para cada i ← 0, i < n, i ← i+1 então
           se temp.valor = valor então
                busca ← i
           temp ← temp.proximo
        busca ← -1
fimBusca
```

2.6. Método de impressão

Para imprimir uma LDDE é necessário, primeiramente, encontrar o ponteiro para o início da lista. A partir de então, é feita a varredura utilizando o ponteiro para o próximo elemento de cada nó para avançar entre os elementos, e percorrer toda a lista, de ponta a ponta. Como a lista possui uma variável de controle de tamanho, representada como n é possível iterar sobre ela. Como a impressão acessa todos os nós da estrutura, seu custo operacional é dado por $O(n)$. O pseudocódigo desse método é representado abaixo:

```
função imprime: escreve
        var temp
        var i: inteiro
        inicio
        temp ← primeiro

        para cada i ← 0, i ← n, i ← i+1 então
           escreve(temp.valor)
```

```
            temp ← temp.proximo
        escreve('\n')

        fimImprime
```

2.7. Pseudocódigo de uma LDDE

```
Algoritmo "LDDE" (Lista Dinamica Duplamente Encadeada)

classe No
    inicio classe
    função construtor
        inicio
        var valor: inteiro
        var proximo, anterior

        valor ← valor
        proximo ← NULL
        anterior ← NULL

        fimConstrutor

classe LDDE
    inicio classe
    função construtor
        inicio
        var primeiro, ultimo
        var n: inteiro

        primeiro ← NULL
        ultimo ← NULL
        n ← 0

        fimConstrutor
```

```
função insere: logico
        var valor: inteiro
        inicio
        var novoNo: instancia de classe
        novoNo ← new No(valor)

        se novoNo = NULL então
            insere ← FALSO

        var noAnterior, noAtual

        noAnterior ← NULL
        noAtual ← primeiro

        enquanto noAtual ≠ NULL e ((noAtual.valor - valor) < 0) então
            noAnterior ← noAtual
            noAtual ← noAtual.proximo

        se noAnterior ≠ NULL então
            noAnterior.proximo ← novoNo
            novoNo.anterior ← noAnterior
        se não
            primeiro ← novoNo

        se noAtual ≠ NULL então
          noAtual.anterior ← novoNo
        se não
          ultimo ← novoNo

        novoNo.proximo ← noAtual
        n ← n+1
        insere ← VERDADEIRO

        fimInsere
```

```
função remove: logico
    var valor, i: inteiro
    var atual, anterior

    inicio
    atual ← primeiro
    anterior ← NULL

    para cada i ← 0, i < n e atual.valor ≠ valor, i ← i+1 então
        anterior ← atual
        atual ← atual.proximo

    se atual = NULL ou atual.valor ≠ valor então
        remove ← FALSO

    se anterior ≠ NULL então
        anterior.proximo ← atual.proximo
    se não
        primeiro ← atual.proximo

    se atual.proximo ≠ NULL então
        atual.proximo.anterior ← atual.anterior
    se não
        ultimo ← anterior

    n ← n-1
    remove ← VERDADEIRO

    fimRemove

função busca: inteiro
    var temp
    var valor, i: inteiro
    inicio
```

```
            temp ← primeiro
            para cada i ← 0, i < n, i ← i+1 então
               se temp.valor = valor então
                    busca ← i
               temp ← temp.proximo
            busca ← -1
        fimBusca

função imprime: escreve
        var temp
        var i: inteiro
        inicio
        temp ← primeiro

        para cada i ← 0, i ← n, i ← i+1 então
           escreve(temp.valor)
           temp ← temp.proximo
        escreve('\n')

        fimImprime
```

2.8. Código de uma LDDE em JavaScript

```
class No{
  constructor(valor){
    this.valor = valor;
    this.proximo = null;
    this.anterior = null;
  }
}

class LDDE{
  constructor(){
    this.primeiro = null;
    this.ultimo = null;
    this.n = 0;
  }

  insere(valor){
    let novoNo = new No(valor);
    if(!novoNo){
      return false;
    }

    let noAnterior = null;
    let noAtual = this.primeiro;

    /**
     * Foi notado um bug no javascript onde décimos, se comparado com unidades retornaria
     * no 'noAtual.valor < valor' o resultado FALSE. Ao invés de transformar os dados somente para Int ou Float,
     * decidiu-se realizar a comparação de outra maneira, igualmente eficiente.
     */
    while(noAtual != null && (noAtual.valor - valor) < 0){
      noAnterior = noAtual;
      noAtual = noAtual.proximo;
    }

    if(noAnterior){ //Caso já possua elementos na lista
      noAnterior.proximo = novoNo;
      novoNo.anterior = noAnterior;
    }
    else{
      this.primeiro = novoNo;
    }

    if(noAtual){
      noAtual.anterior = novoNo}
    else{
      this.ultimo = novoNo;
    }

    novoNo.proximo = noAtual;
    this.n++;
    return true;
  }

  busca(valor){
        let temp = this.primeiro;
        for(let i=0; i < this.n; i++) {
            if (temp.valor == valor)
                return i;
            temp = temp.proximo;
        }
        return -1;
  }

  remove(valor){
    let atual = this.primeiro;
    let anterior = null;

    for(let i=0; i < this.n && atual.valor != valor; i++) {
```

```
68      for(let i=0; i < this.n && atual.valor != valor; i++) {
69          anterior = atual;
70          atual = atual.proximo;
71      }
72      if(!atual || atual.valor != valor){
73          return [false,'O valor a ser removido não existe.'];
74      }
75
76      if(anterior){
77          anterior.proximo = atual.proximo;
78      }
79      else{
80          this.primeiro = atual.proximo;
81      }
82
83      if(atual.proximo){
84          atual.proximo.anterior = atual.anterior;
85      }
86      else{
87          this.ultimo = anterior;
88      }
89      this.n--;
90      return [true,'Valor removido com sucesso'];
91    }
92
93    imprime(){
94      let temp = this.primeiro;
95      for(var i = 0; i < this.n; i++){
96        console.log(temp.valor);
97        temp = temp.proximo;
98      }
99      console.log("\n");
100   }
101 }
```