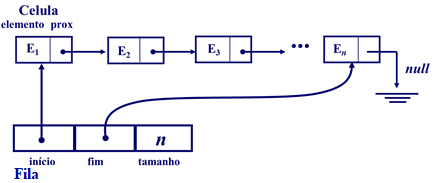
#### *Listas e Pilhas – implementação por Encadeamento*

**Implementação do *TAD Fila* por encadeamento**

Sendo uma estrutura linear, unidimensional, uma *fila* pode ser adequadamene implementada por encadeamento. A implementação de uma *Fila*, por encadeamento simples, é muito similar à da *Lista* por encadeamento simples. Os elementos da *Fila* são encadeados via apontadores. A figura a seguir ilustra a semelhança da estrutura de dados da *Fila* e da *Lista*.



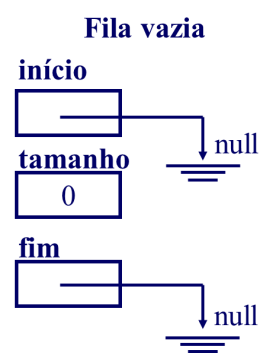
A abstração acima pode facilmente ser traduzida para linguagens computacionais. A seguir, sua implementação em Português Estruturado e em C:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| Elemento: **Registro**  ***// campos da estrutura de dados da Lista***  **FimRegistro**  Celula: **Registro**  elemento: Elemento  prox: ↑Celula  **FimRegistro**  Fila: **Registro**  inicio, fim: ↑Celula  tamanho: Inteiro  **FimRegistro** | **typedef struct {**  ***// campos da estrutura de dados da Lista***  **} Elemento;**  **typedef struct** Cel \* **Ponteiro;**  **typedef struct** Cel **{**  **Elemento** elemento;  **Ponteiro** prox;  **} Celula;**  **typedef struct {**  **Ponteiro** inicio, fim;  **int** tamanho;  **} Fila;** |

As operações da Fila implementadas neste estudo são as mesmas consideradas na implementação de *Fila por Arranjo Circular*:

* *iniciaFila (Fila \*fila)*
* *isEmpty(Fila \*fila);*
* *enqueue(Elemento x, Fila \*fila);*
* *dequeue(Elemento \*x, Fila \*fila);*

A operação *iniciaFila* faz os ponteiros *inicio* e fim da *fila* apontarem para *null* e *tamanho* igual a zero, conforme ilustrado a seguir:



A codificação dessa operação:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** iniciaFila (**var** fila: Fila)  **Inicio**  fila↑inicio 🡨 null  fila↑fim 🡨 null  fila↑tamanho 🡨 0  **FimProcedimento** | **void** iniciaFila (**Fila** \*fila)  **{**  fila->inicio = NULL;  fila->fim = fila->inicio;  fila->tamanho = 0;  **}** |

A operação *isEmpty* é *booleana* e verifica se o *inicio* é igual a *null*.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** isEmpty (**var** fila: Fila): boolean  **Inicio**  **Retorne** (fila**↑**inicio = null)  **FimFuncao** | **int** isEmpty (**Fila** \*fila)  **{**  **return** (fila->inicio == NULL);  **}** |

A operação *enqueue* insere o elemento ***x***, recebido como parâmetro, no final da *fila*. Esta operação possui o mesmo algoritmo da operação *insereFinal* da *Lista Encadeada*. Detalhes dos 6 passos deste algoritmo podem ser recordados na apostila sobre implementação da *Lista Encadeada*.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** insereFinal(x:Elemento, **var** lst:Lista)  **Var** pAux: ↑Celula  **Inicio**  pAux 🡨 Aloc (↑Celula)  pAux↑elemento 🡨 x  pAux↑prox 🡨 null  **Se** (vazia(lst)) **então**  lst↑inicio 🡨 pAux  **Senao**  lst↑fim↑prox 🡨 pAux  **FimSe**  lst↑fim 🡨 pAux  lst↑tamanho 🡨 lst↑tamanho  **FimProcedimento** | void enqueue (Elemento x, Fila \*fila)  {  Ponteiro pAux;  pAux = (Ponteiro)malloc(sizeof (Celula));  pAux->elemento = x;  pAux->prox = NULL;  if (isEmpty(fila))  fila->inicio = pAux;  else  fila->fim->prox = pAux;  fila->fim = pAux;  fila->tamanho++;  } |

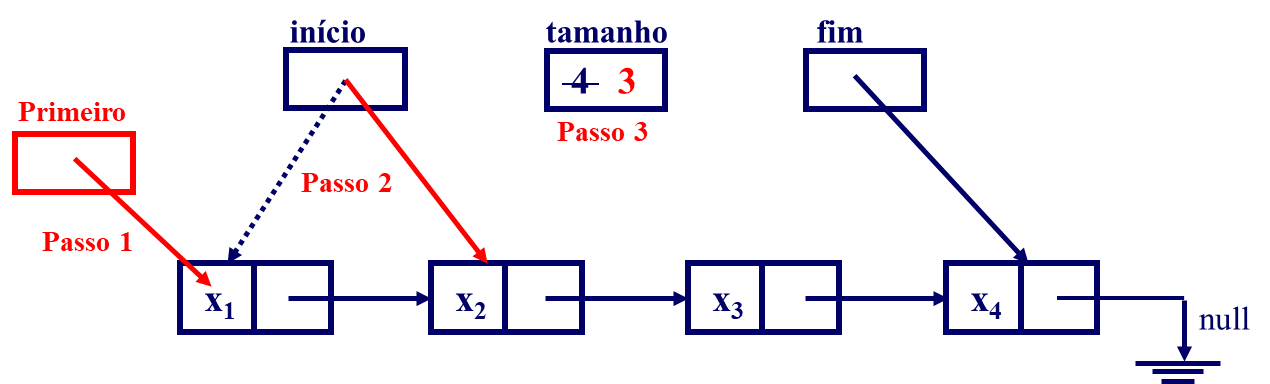
A operação *dequeue* retira o elemento do inicio da fila e retorna este para o chamador. Testa inicialmente se fila está vazia e, neste caso, retorna código de erro adequado. Se fila não vazia, a retirada do elemento do inicio da fila segue os seguintes passos:

Passo 0: Verificar *underflow* e caso afirmativo interromper execução e retornar *código de erro*;

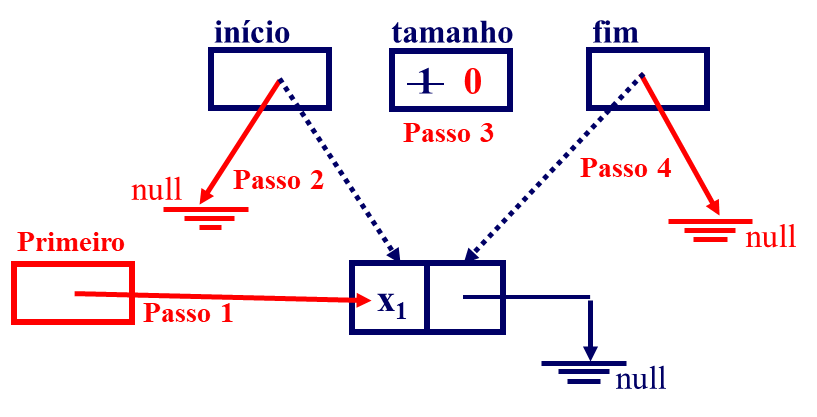
Passo 1: Salvar a referência do ***elemento*** do início da *fila*;

Passo 2: Atribuir ao *início* a referência do *próximo*;

Passo 3: Decrementar o tamanho da fila.



Passo 4: Caso a retirada seja do único (último) elemento, ajustar a referência *fim.*



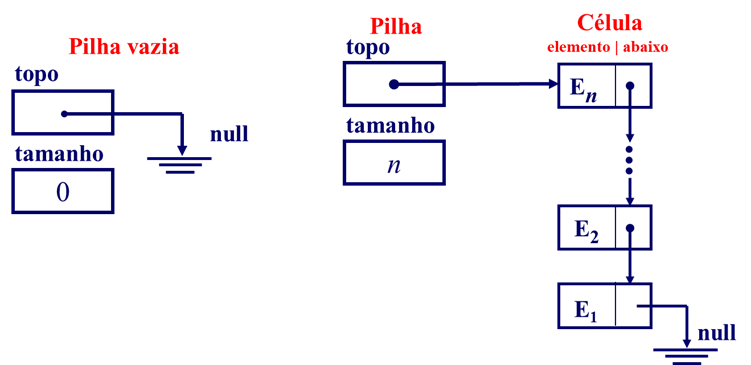
Segue a codificação da operação dequeue:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** dequeue (**var** x: Elemento,  **var** fila: Fila): inteiro  **Var** primeiro: ↑Celula  **Inicio**  **Se** ( isEmpty(fila)) **então**  **Retorne** (999)  **FimSe**  primeiro🡨 fila↑inicio  x = primeiro↑elemento  fila↑inicio 🡨 fila↑inicio↑prox  fila↑tamanho 🡨 l↑tamanho – 1  **Se** (fila↑fim = primeiro) **então**  fila↑fim 🡨 null  **FimSe**  LiberaMemoria(primeiro)  **Retorne** (0)  **FimFuncao** | **int dequeue**(**Elemento** \*x, **Fila** \*fila)  **{**  if (isEmpty(fila)) *//Passo 0*  return 999; *// Código para fila vazia*  **Ponteiro** primeiro = fila->inicio; *//Passo 1*  \*x = primeiro->elemento;  fila->inicio = fila->inicio->prox; *//Passo 2*  fila->tamanho--; *//Passo 3*  if(fila->fim == primeiro) *//Passo 4*  fila->fim = NULL;  free (primeiro);  return 0; *// Desempilhou com sucesso*  **}** |

**Implementação do *TAD Pilha* por encadeamento**

A implementação do *TAD Pilha* será por meio de um **trabalho prático**, que o aluno deverá desenvolver, valendo nota de participação e um bônus de até 0.5 ponto extra na média do bimestre, se o trabalho for exclusivo (sem evidência de cópia por outro aluno ou grupo) e se estiver correto.

A *pilha*, assim como a *fila*, possui estrutura exatamente igual a *lista*. A figura abaixo abstrai uma *pilha*. Note que, diferentemente da lista e fila, a pilha precisa de apenas uma marca para controle da estrutura: o ***topo***.



Assim, as estruturas de dados necessárias são Pilha e Celula, ilustradas a seguir:

As operações do TAD Pilha que serão consideradas são:

1. ***criaPilha***: inicializa uma *pilha* com *topo = null* e *tamanho = 0*;
2. ***isEmpty***: operação *booleana* que verifica se a pilha está vazia (*topo == null*);
3. ***push***: insere um elemento no topo da *pilha* (empilha);
4. ***pop***: retira o elemento do topo da *pilha* (desempilha);
5. ***peek***: espia (ou consulta) o elemento do topo da *pilha* (sem desempilhar).

A operação ***push*** tem o seguinte comportamento:

**Passo 1**: Salvar a referencia do *topo* em uma variável *auxiliar*;

**Passo 2**: Alocar nova celula armazenado sua referencia em *topo*;

**Passo 3**: Colocar na célula criada o elemento empilhado;

**Passo 4**: Atribuir a *abaixo* a referencia guardada na variável auxiliar;

**Passo 5**: Incrementar o tamanho da pilha.

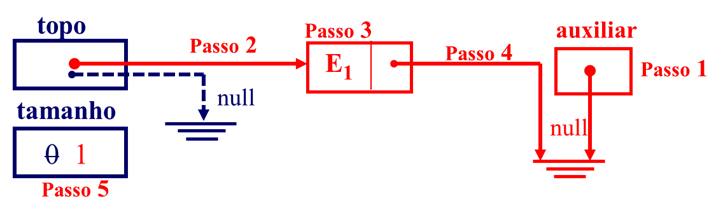


Ilustração do algoritmo de inserção na pilha vazia

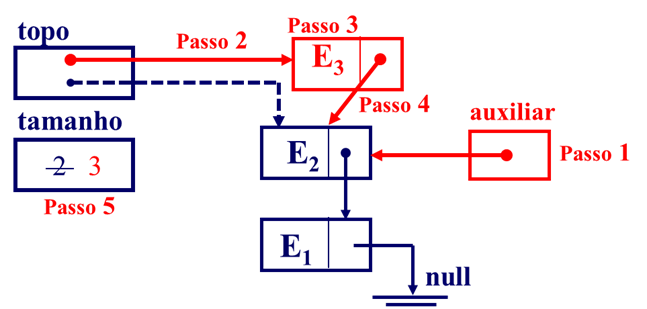


Ilustração do algoritmo de inserção na pilha não vazia

A operação ***pop*** tem o seguinte comportamento:

**Passo 0**: Verificar *underflow* e caso afirmativo interromper execução e retornar *código de erro*;

**Passo 1**: Salvar a referência do *elemento* do *topo* da pilha;

**Passo 2**: Atribuir ao *topo* a referência *abaixo*;

**Passo 3**: Decrementar o *tamanho* da *pilha*;

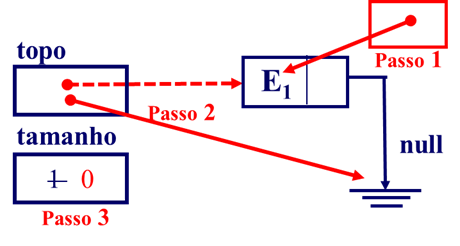


Ilustração do algoritmo de retirada do último elemento da pilha

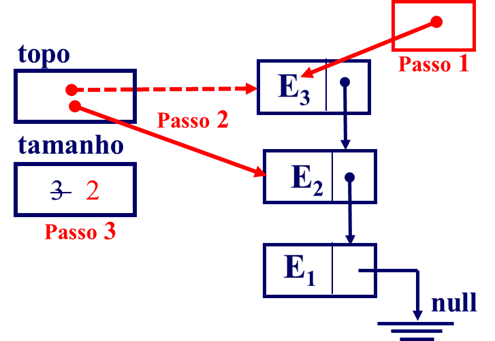


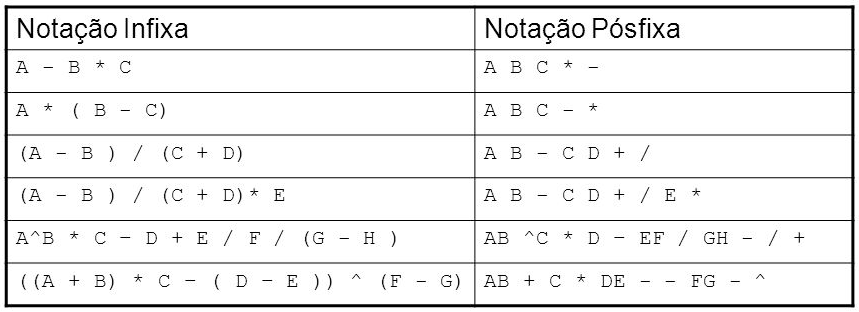
Ilustração do algoritmo de retirada de elemento não último da pilha

A operação ***peek*** apenas devolve o elemento do *topo* da *pilha*, sem retirá-lo. Neste caso, o algoritmo é o mesmo do *pop*, sem os passos 2 e 3.

**Especificação do Trabalho Prático**

O aluno ou grupo deverá implementar na linguagem C um ***TAD Pilha***, baseado nos algoritmos descritos acima e se espelhando no *TAD Fila* já demonstrado em prova de conceito.

Para testar se o TAD Pilha implementado esta funcionando, implementar o seguinte trabalho: desenvolver uma aplicação para resolver uma expressão na notação ***Polonesa Reversa*** ou ***Pós-Fixada***ou***Posfixa***. Nesta notação, os operandos precedem os operadores, como nos exemplos do quadro abaixo.



O interessante desta notação é que as precedências das operações ficam implícitas na expressão, bastando, sempre que encontrar um operador, executar a operação respectiva com os dois últimos operandos, gerando um novo operando.

Por facilidade e visando a atender aos objetivos da disciplina ED, serão feitas as seguintes simplificações nesta solução:

1. Assume-se que a entrada da expressão será na notação *pósfixa*. O algoritmo de mapeamento da expressão *infixa* para *pósfixa* (como os compiladores geralmente fazem) não será objeto deste trabalho, mas pode ser facilmente pesquisado nas bibliografias ou em publicações na Internet.
2. Também para evitar detalhes de *parsing* da expressão (tarefa típica de compiladores fora do escopo deste trabalho), assume-se que as expressões de entrada terão **operandos** com um único dígito (valores entre 0 e 9) e apenas os operadores **+ - \*** e **/** para as operações soma, subtração, multiplicação e divisão real, respectivamente. Assim, como exemplo, a *string* de entrada:

85-93+/

é válida e representa a expressão (8 – 5)/(9+3), cujo resultado deve ser 0,25.

Considerando as premissas acima, o sistema terá como **entrada** uma string de até 70 caracteres, cujo conteúdo será a expressão na notação pós-fixada, fornecida pelo usuário. O conteúdo da entrada deve ser operandos, operadores e separadores. Os separadores serão espaços em branco, como apresentado no exemplo abaixo. Após entrada da expressão, o sistema “varre” a string e aplica o seguinte algoritmo, considerando uma pilha inicialmente vazia:

1. Despreza separadores e empilha **operandos,**  até encontrar um **operador**;
2. ao encontrar um **operador**, desempilha dois **operandos,** executa a operação, e empilha o resultado (que se torna um operando);

Ao final da varredura, a pilha deverá conter apenas um operando, que será o resultado da expressão. Senão expressão inválida. O exemplo abaixo ilustra o algoritmo, onde **V** representa o **V**alor final da expressão.

**Exemplo:** A B / D E \* + A –

Algoritmo:

encontrando **operando**, empilha;

encontrando **operador**, desempilha dois operados, executa a operação e empilha o resultado;

