Universidade de Brasília – Instituto de Ciências Exatas
Departamento de Ciências da Computação

Estruturas de Dados – Trabalho 1

Avaliação de Expressões Ariméticas (Forma Posfixa)

Lucas de Almeida Abreu Faria e Mateus Berardo de Souza Terra
Prof. Eduardo A. P. Alchieri
30 de Março de 2018

1 Introdução

O trabalho 1 proposto para a matéria de Estruturas de Dados consiste na implementação de uma calculadora de expressões aritméticas posfixas a partir de entradas infixas em C, utilizando principalmente pilhas.

Para tanto, o programa a ser desenvolvido receberá como entrada, por meio da entrada padrão e da função *scanf* uma expressão na forma infixa, ou seja, com os operadores entre os operandos, conforme os exemplos a seguir:

*A+B*C*
(A+B)/C
*(A/(C-D))*B*

O programa fará, então, uma validação da *string* de entrada, avaliando se os escopos abertos por parênteses estão devidamente fechados. O algoritmo para tal fará uso de uma pilha e está detalhado no capítulo sobre a implementação. Depois de validada, a expressão será transformada para a forma posfixa equivalente, como nos exemplos a seguir, referentes aos modelos já apresentados:

ABC+*
AB+C/
*ACD-/B**

A partir da expressão posfixa, o resultado será calculado por meio de um algoritmo baseado em uma pilha de cálculo e este será exibido na tela por meio da saída padrão, utilizando a função *printf*.

Para favorecer a interação com o usuário, foi desenvolvida uma interface em **Python 3**, utilizando o *port* do framework **Qt** para esta linguagem. Essa GUI recebe a expressão por meio de um teclado virtual, ou do teclado físico, e

ao receber o sinal de igual (‘=’) na entrada, inicia o programa em **C** e, por meio de um *Pipe* direciona a expressão para ele, que retorna pelo *Pipe* o resultado que deve ser exibido na tela

2 Implementação

Dividiremos a implentação do programa em tópicos, a saber:

I – Estruturas de Dados
II – Validação e Processamento de Entrada Infixa
III – Processamento de Expressão Posfixa e Saída
IV – Interface de Usuário

Além do detalhamento das implementações, faremos também uma rápida análise da complexidade utilizando a notação ***O***. Para a análise, consideraremos o custo de instruções da biblioteca padrão do **C** e declarações com custo igual a 1, bem como comparações.

I – Estruturas de Dados

As estruturas de dados aplicadas a resolução do problema foram pilhas e filas, implementadas sobre uma lista encadeada. A seguir detalharemos esses conceitos e sua implementação em nosso programa.

Lista Encadeada: “*Uma lista é uma estrutura de dados dinâmica. O número de nós de uma lista pode variar consideravelmente à medida que são inseridos e removidos elementos. A natureza dinâmica de uma lista pode ser comparada à natureza estática de um vetor, cujo tamanho permanece constante. (...) Cada item na lista é chamado nó e contém dois campos, um campo de informação e um campo do endereço seguinte. O campo de informação armazena o real elemento da lista. O campo do endereço seguinte contém o endereço do próximo nó na lista.*”. Abaixo mostramos uma representação esquemática da lista e de sua disposição na memória.

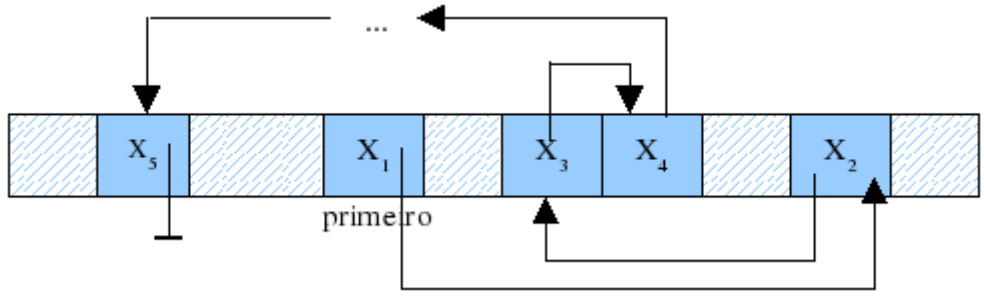

Figura 1: Disposição de uma lista encadeada na memória

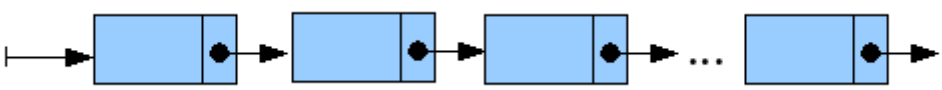

Figura 2: Representação esquemática de uma lista simplesmente encadeada

Para facilitar a interpretação dos dados das expressões aritméticas, que são formadas por números e caracteres, optamos por implementar uma lista heterogênea, ou seja, em que seja possível armazenar diferentes tipos de dados. Para tanto, os elementos foram declarados da seguinte forma:

```
typedef struct elemento{
        void *dado;
        int tipoDado;
                struct  elemento
*proximo;
} t_element;
```

Para acomodar dados de diferentes tipos, utilizamos os endereços desses dados, que foram armazenados como ponteiros para *void*. Para facilitar a posterior interpretação desses dados, adicionamos um campo inteiro que identifica o tipo a ser utilizado para obter um valor útil. Duas constantes foram definidas para diferenciar os 2 tipos utilizados no programa: *t_char* e *t_double.* O terceiro campo do elemento é um ponteiro para outro elemento, o seguinte na lista.

A lista também é uma *struct*, que possui dois ponteiros, um para seu primeiro elemento e outro para o seu último elemento.

```
typedef struct list{
        t_element *inicio;
        t_element *fim;
} t_list;
```

As funções implementadas foram as seguintes:

*t_list *newList()* → Retorna um ponteiro para lista com os campos inicio e fim nulos. Não recebe nenhum argumento. Cria uma nova lista vazia. Possui uma complexidade ***O(1)***, pois não tem laços de repetição;

*void addStart(t_list *list, void *dado, int tipoDado)*→ Não possui retorno. Recebe como argumentos um ponteiro para a lista que receberá o novo elemento, o ponteiro para o dado a ser inserido e o tipo do dado. Insere um novo elemento no início da lista. Possui uma complexidade ***O(1)***, pois não possui laços de repetição;

*void addEnd(t_list *list, void *dado, int tipoDado)* → Não possui retorno. Recebe como argumentos um ponteiro para a lista que receberá o novo elemento, o ponteiro para o dado a ser inserido e o tipo do dado. Insere um novo elemento no final da lista. Possui uma complexidade ***O(1)***, pois não possui laços de repetição ;

*int isEmpty(t_list *list)* → Retorna um inteiro. Recebe uma lista como argumento. Verifica se o início da lista é nula, ou seja, se não há elementos, retornando 1 caso esteja vazia e 0 caso contrário. Possui uma complexidade ***O(1)***, pois não possui laços de repetição ;

*char removeFirstChar(t_list *list)* → Retorna um caractere. Recebe uma lista como argumento. Retorna o conteúdo armazenado no endereço do primeiro elemento da lista interpretado como caractere. Também libera os espaços alocados para as informações e para o elemento. Possui uma complexidade ***O(1)***, pois não possui laços de repetição;

*double removeFirstDouble(t_list *list)* → Retorna um valor de ponto flutuante de precisão dupla (*double)*. Recebe uma lista como argumento. Retorna o conteúdo armazenado no endereço do primeiro elemento da lista interpretado como *double*. Também libera os espaços alocados para as informações e para o elemento. Possui uma complexidade ***O(1)***, pois não possui laços de repetição;

*int getNextType(t_list *list)* → Retorna um inteiro. Recebe uma lista como argumento. Lê o tipo de dado do primeiro elemento da lista. Possui uma complexidade ***O(1)***, pois não possui laços de repetição.

Pilha: "*Uma pilha é um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados itens em uma extremidade chamada topo da pilha*". Uma pilha é como uma pilha de pratos, para acessar um prato que esteja no meio, é necessário desempilhar os que estiverem acima dele. Abaixo podemos ver uma figura que ilustra essa estrutura de dados.

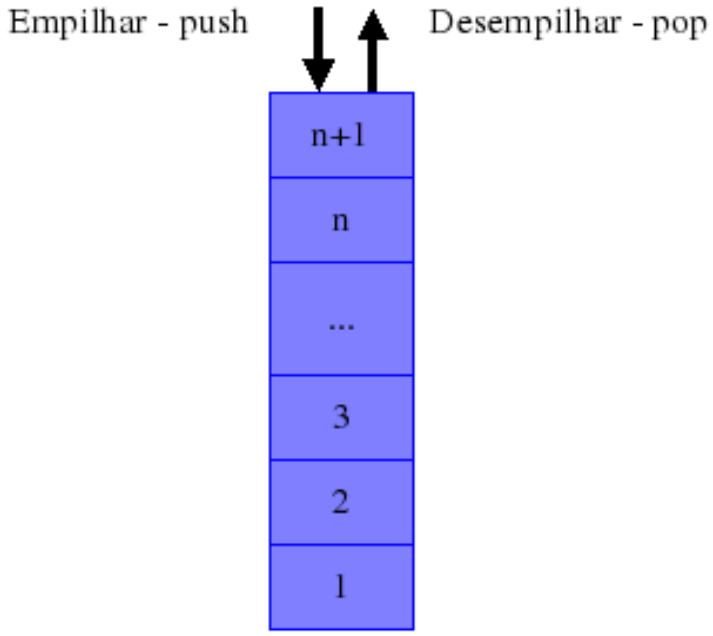

Figura 3: Pilha de dados

Em nosso programa foi implementada baseada em uma lista encadeada com acesso apenas ao início, seja pra inserção ou remoção. Como foi baseada em nossa implementação de lista descrita acima, também pode ser heterogênea. A pilha é uma *struct*, definida a seguir:

```
typedef struct stack{
```

```
        t_list *l;
} t_stack;
```

Para sua manipulação, foram desenvolvidas as seguintes funções:

*t_stack *newStack()* → Retorna um ponteiro para pilha. Não recebe argumentos. Inicializa uma pilha com uma nova lista. Possui uma complexidade ***O(1)***, pois não possui laços de repetição;

*void push(t_stack *stack, void *dado, int tipoDado)* → Não possui retorno. Recebe como argumentos um ponteiro para a pilha que receberá o novo elemento, o ponteiro para o dado a ser inserido e o tipo do dado. Insere um novo elemento no topo da lista utilizando a função *addStart.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo;

*char popChar(t_stack *stack)* → Retorna um caractere. Recebe uma pilha de argumento. Obtém a informação do primeiro elemento da lista, ou seja, do topo da pilha, por meio da função *removeFirstChar.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo*;*

*double popDouble(t_stack *stack)* → Retorna um valor de ponto flutuante com precisão dupla. Recebe uma pilha de argumento. Obtém a informação do primeiro elemento da lista, ou seja, do topo da pilha, por meio da função *removeFirstDouble.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo*;*

*int isStackEmpty(t_stack *stack)* → Retorna um inteiro. Recebe uma pilha de argumento. Verifica se a pilha está vazia por meio da função *isEmpty.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo*;*

*int getNextStackType(t_stack *stack)* → Retorna um inteiro. Recebe uma pilha de argumento. Verifica o tipo de dado armazenado no topo da pilha por meio da função *getNextType.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo.

Fila: “*Uma fila é um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada início da fila) e no qual podem-se inserir itens na outra extremidade (chamada final da fila). (...) O primeiro elemento inserido numa fila é o primeiro a ser removido. Por essa razão, uma fila é ocasionalmente chamada lista* ***fifo*** *(first-in, first-out — o primeiro que entra é o primeiro a sair), ao contrário de uma pilha, que é uma lista* ***lifo*** *(last-in, first-out — o último a entrar é o primeiro a sair).*”. Uma ótima analogia seria uma fila de banco, onde o primeiro a chegar, será o primeiro a ser atendido. Abaixo temos uma imagem esquemática que demonstra o funcionamento desta estrutura.

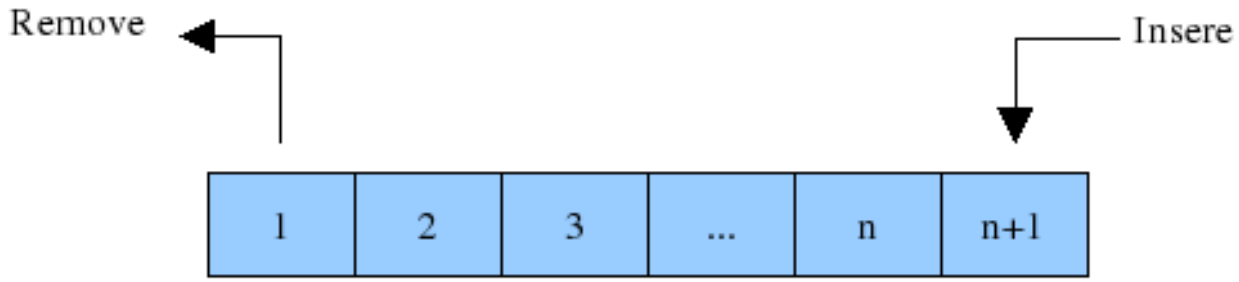

Figura 4: Fila de dados

Em nosso programa foi implementada baseada em uma lista encadeada com inserção no final e remoção no início, para diminuir a complexidade destas operações. Como foi baseada em nossa implementação de lista, também pode ser heterogênea. A fila é uma *struct,* definida a seguir:

```
typedef struct queue{
        t_list *l;
} t_queue;
```

Para sua manipulação, foram implementadas as seguintes funções:

*t_queue *newQueue()* → Retorna um ponteiro para fila. Não recebe argumentos. Inicializa uma fila com uma nova lista. Possui uma complexidade ***O(1)***, pois não possui laços de repetição ;

*void add(t_queue *queue, void *dado, int tipoDado)* → Não possui retorno. Recebe como argumentos um ponteiro para a fila que receberá o novo elemento, o ponteiro para o dado a ser inserido e o tipo do dado. Insere um novo elemento no final da lista utilizando a função *addEnd.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo;

*char removeChar(t_queue *queue)* → Retorna um caractere. Recebe uma fila de argumento. Obtém a informação do primeiro elemento da lista, ou seja, do início da fila, por meio da função *removeFirstChar.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo;

*double removeDouble(t_queue *queue)* → Retorna um valor de ponto flutuante com precisão dupla. Recebe uma fila de argumento. Obtém a informação do primeiro elemento da lista, ou seja, do início da fila, por meio da função *removeFirstDouble.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo;

*int isEmptyQueue(t_queue *queue)* → Retorna um inteiro. Recebe uma fila de argumento. Verifica se a fila está vazia por meio da função *isEmpty.*

Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo;

*int getNextQueueType(t_queue *queue)* → Retorna um inteiro. Recebe uma fila de argumento. Verifica o tipo de dado armazenado no início da fila por meio da função *getNextType.* Possui uma complexidade ***O(1)***, pois apenas chama uma função com esse custo.

Essas foram as estruturas de dados utilizadas na implementação do programa. Todas foram implementadas de forma a serem reutilizáveis, ou seja, como bibliotecas.

II – Validação e Processamento de Entrada Infixa

A partir do uso dessas estruturas, foi desenvolvida a lógica do programa, começando com o recebimento da expressão de entrada, validação da mesma e transformação em expressão posfixa.

Algumas funções de comparação foram implementadas para facilitar o tratamento dos caracteres em longo de várias partes do código. São elas:

*int isNumber(char * dado)* → Recebe um ponteiro para o caractere que será analisado. Retorna 1 se o dado for um dígito, retorna 0 caso contrário. Possui complexidade ***O(1)***, pois tem custo sempre constante independentemente do argumento.

*int isOperator(char * dado)* → Recebe um ponteiro para o caractere que será analisado. Retorna 1 se o dado for um operador, retorna 0 caso contrário. Possui complexidade ***O(1)***, pois tem custo sempre constante independentemente do argumento.

*int isParenthesis(char * dado)* → Recebe um ponteiro para o caractere que será analisado. Retorna 1 se o dado for igual a ‘(‘ ou ‘)’, 0 caso contrário. Possui complexidade ***O(1)***, pois tem custo sempre constante independentemente do argumento.

*int isLowerPrior(char * dado)* → Recebe um ponteiro para o caractere que será analisado. Retorna 1 se o dado for um operador de baixa prioridade (‘+’ ou ‘-’), retorna 0 caso contrário. Possui complexidade ***O(1)***, pois tem custo sempre constante independentemente do argumento.

A leitura da expressão de entrada é feita segundo o algoritmo abaixo:

```
void getExpression(t_queue * queue){
	char dado = '\0';
	double number = 0.0;

	while(dado != '\n'){
		scanf("%c", &dado);
		while(isNumber(&dado)){
			number = (number * 10.0) + (dado - '0');
			scanf("%c", &dado);
```

```
                    if(!isNumber(&dado)){
                        add(queue, &number, t_double);
                        number = 0.0;
                    }
                }
                if(dado != ' ' && dado != '\n')
                    add(queue, &dado, t_char);
        }
}
```

A função *getExpression* recebe como argumento um ponteiro para fila, na qual a expressão de entrada será armazenada. A leitura da entrada padrão é feita de forma que seja recebido um caractere por vez, armazenando-o na variável *dado,* até que o valor lido seja '\n', indicando o fim da expressão. Se o dado for um dígito, ele é subtraído do caractere '0' para que obtenhamos seu valor numérico, e somado com o valor anterior da variável *number* multiplicado por 10, para tratar números com mais de um algarismo. Lê-se o próximo caractere: caso ele não seja um dígito, significa que os algarismos do número acabaram, e o valor armazenado em *number* é enfileirado. Caso contrário, repete-se o algoritmo. Se o dado não for um dígito nem os caracteres ' ' e '\n', ele é enfileirado. A complexidade dessa função é ***O(n)*** devido ao laço de repetição que é executado ***n*** vezes, sendo ***n*** o número de caracteres da expressão de entrada.

Após isso, a fila que contém a expressão a ser avaliada é submetida a um processo de validação, no qual adotou-se os seguintes critérios:

Se o elemento que acabou de ser retirado da fila é do tipo double, a expressão é inválida caso:

→ O elemento anterior também seja do tipo double;
→ O elemento anterior seja igual a '('.
→ Isso invalida expressões como "A+B C" e "(A+B)C".
→ A função que realiza esses testes é:

int okTestsDouble(int lastType, char lastChar) → Recebe o tipo de dado do elemento anterior retirado da fila e o último caractere retirado da fila. A função retorna 1 caso os testes tenham tido resultados válidos, e 0 caso contrário. A função possui complexidade ***O(1)*** pois seu custo é sempre constante, independentemente dos argumentos passados.

Se o elemento que acabou de ser retirado da fila é igual a '(', a expressão é inválida caso:

→ O elemento anterior seja do tipo double;
→ O elemento anterior seja igual a ')'.
→ Isso invalida expressões como "A(B+C)" e "(A*B)(C/D)".

Se o elemento que acabou de ser retirado da fila é igual a ')', a expressão é inválida caso:

→ O elemento anterior seja um operador.
→ Isso invalida expressões como "(A-B*)"

Se o elemento que acabou de ser retirado da fila é um operador, a expressão é inválida caso:

→ O elemento anterior também seja um operador;

→ O elemento anterior é igual a ‘(‘ e o que acabou de ser retirado é um operador de alta prioridade (* ou /).

→ Isso invalida expressões como “A++B” e “(*A-B)”

→ A função que testa todos esses outros casos é:

int okTestsChar(char dado, int lastType, char lastChar) → Recebe o dado do elemento que acabou de ser retirado, o tipo de dado do elemento anterior retirado da fila e o último caractere retirado da fila. A função retorna 1 caso os testes tenham tido resultados válidos, e 0 caso contrário. A função possui complexidade ***O(1)*** pois seu custo é sempre constante, independentemente dos argumentos passados.

Também foram consideradas inválidas expressões que:

→ Comecem com os operadores ‘*’ ou ‘/’;

→ Possuam caracteres diferentes de dígitos, operadores e parênteses;

→ Possuam um número de operadores maior ou igual ao número de elementos do tipo double;

→ Tenham números diferentes de ‘(‘ e ‘)’;

→ Existam ‘)’ que não fecham nenhum ‘)’.

A validação foi feita pela seguinte função:

```
int validExpression(t_queue * queue, t_queue * validada){
	t_stack * stack = newStack();
	char lastChar = '\0', dadoChar;
	int tipoDado, lastType = -1, opCount = 0, doubleCount = 0;
	double dadoDouble;

	while(!isEmptyQueue(queue)){
		tipoDado = getNextQueueType(queue);
		if(tipoDado == t_double){
			dadoDouble = removeDouble(queue);
			if(!okTestsDouble(lastType, lastChar))
				return 0;
			doubleCount++;
			lastType = t_double;
			add(validada, &dadoDouble, t_double);
		}else{
			dadoChar = removeChar(queue);
			if(!okTestsChar(dadoChar, lastType, lastChar))
				return 0;
			if(dadoChar == '(')
				push(stack, &dadoChar, t_char);
			else if(dadoChar == ')'){
				if(isStackEmpty(stack))
					return 0;
				if(popChar(stack) != '(')
```

```
                                return 0;
                        }else if(isLowerPrior(&dadoChar)){
                                if((lastChar == '(' && lastType == t_char) || lastType
== -1){
                                        dadoDouble = 0;
                                        doubleCount++;
                                        add(validada, &dadoDouble, t_double);
                                }
                        }
                        if(isOperator(&dadoChar))
                                opCount++;
                        lastType = t_char;
                        lastChar = dadoChar;
                        add(validada, &dadoChar, t_char);
                }
        }
        if(opCount >= doubleCount)
                return 0;
        if(!isStackEmpty(stack))
                return 0;
        return 1;
}
```

A função *validExpression* recebe como argumentos um ponteiro para a fila que contém a expressão que será validada, e um ponteiro para uma outra fila, que na qual os elementos irão sendo armazenados conforme a expressão for sendo validada. A validação é feita da seguinte forma: enquanto a fila com a expressão de entrada não estiver vazia, lê-se o tipo de dado do primeiro elemento da fila, e este é retirado conforme o tipo lido. Se for do tipo double, realizam-se os testes com a função *okTestsDouble.* Caso os testes sejam válidos, marca-se que o último tipo de dado é t_double e o número é adicionado à fila de saída. Se o elemento retirado for do tipo char, realizam-se os testes com a função *okTestsChar.* Caso os testes sejam válidos, prossegue-se da seguinte forma: se o elemento retirado for um ‘(‘, o mesmo é empilhado. Se o elemento retirado for um ‘)’, checa-se se a pilha, se esta estiver vazia, a expressão é inválida, e caso contrário, desempilha-se um elemento e checa-se se o mesmo é um ‘(‘, para que a expressão seja válida. Se o elemento retirado for um operador de baixa prioridade que vem no início da expressão ou logo após um parênteses, insere-se um 0 na fila de saída para que o cálculo seja realizado de forma correta posteriormente. Então, atualiza-se a variável *lastChar* com o valor do elemento retirado, e marca-se que o último tipo de dado é t_char. Por fim, o elemento retirado é adicionado à fila de saída. Depois de retirar todos os elementos da fila de entrada, checa-se se o número de operadores é menor que o número de operandos, e se a pilha está vazia, condições necessárias para que a expressão seja válida. A função retorna 1 caso a expressão seja válida, e 0 caso contrário. Sua complexidade é ***O(n)*** devido ao laço de repetição que é executado ***n*** vezes, sendo ***n*** o número de elementos da fila de entrada.

Após o processo de validação, a fila que contém a expressão validada passa pelo processo transformação em expressão posfixa, que é realizado pela função a seguir:

```
t_queue * transform(t_queue * queue){
	t_stack * stack = newStack();
	t_queue * saida = newQueue();
	int tipoDado;
	double dadoDouble;
	char dadoChar;

	while(!isEmptyQueue(queue)){
		tipoDado = getNextQueueType(queue);
		if(tipoDado == t_double){
			dadoDouble = removeDouble(queue);
			add(saida, &dadoDouble, t_double);
		}else{
			dadoChar = removeChar(queue);
			if(isParenthesis(&dadoChar))
				parenthesisProcess(&dadoChar, stack, saida);
			else
				operatorProcess(&dadoChar, stack, saida);
		}
	}
	while(!isStackEmpty(stack)){
		dadoChar = popChar(stack);
		add(saida, &dadoChar, t_char);
	}
	free(queue);
	return saida;
}
```

A função transform recebe um ponteiro para a fila que contém a expressão a ser transformada, e inicializa uma fila de saída e uma pilha vazias. Enquanto a fila com a expressão não estiver vazia, lê-se o tipo de dado do próximo elemento. Caso ele seja do tipo double, o elemento é removido e seu dado é adicionado à fila de saída. Caso ele seja do tipo char e for um parênteses, a função *parenthesisProcess* é chamada:

void parenthesisProcess(char * dado, t_stack * stack, t_queue * saida) → Recebe um ponteiro para o dado, um ponteiro para a pilha e um ponteiro para a fila de saida. Se o dado for um ‘(‘, ele é empilhado. Caso seja um ‘)’, desemplilham-se os elementos da pilha, que são adicionados à fila de saída, até que o dado do elemento desempilhado seja um ‘(‘. A complexidade dessa função é ***O(n)*** devido ao laço de repetição que é executado **n** vezes, sendo **n** o número de elementos na pilha antes do ‘(‘ mais próximo.

Continuando na função *transform,* caso o dado do elemento retirado da fila seja um operador, a função *operatorProcess* é chamada:

void operatorProcess(char * dado, t_stack * stack, t_queue * saida) → Recebe um ponteiro para o dado, um ponteiro para a pilha e um ponteiro para a fila de saida. Enquanto a pilha não estiver vazia e seu primeiro elemento for um operador com prioridade maior ou igual ao dado recebido como argumento, esse operador é desempilhado e adicionado à fila de saída. Após o laço terminar, o operador recebido como argumento é empilhado. A complexidade dessa função é ***O(n)*** devido ao laço de repetição que é executado **n** vezes, sendo **n** o número de elementos na pilha que deverão ser desempilhados até que o operador encontrado possa ser empilhado.

Por fim, a função *transform* adiciona os elementos restantes da pilha à fila, à medidada que são desempilhados, e retorna um ponteiro para a fila de saída. Essa função possui complexidade ***O(n)*** devido ao laço de repetição que é executado **n** vezes, sendo **n** o número de elementos da fila de entrada.

III – Processamento de Expressão Posfixa e Saída

A transformação gera uma fila, equivalente a expressão posfixa da entrada. A partir dessa fila, será calculado o valor resultante seguindo o algoritmo abaixo:

```
double calculate(t_queue *queue){
   t_stack *stack = newStack();
   double op1, op2, resultado;
   while(!isEmptyQueue(queue)){
      if(getNextQueueType(queue) == t_double){
         double dado = removeDouble(queue);
         push(stack, &dado, t_double);
      } else {
         char operacao = removeChar(queue);
         op1 = popDouble(stack);
         op2 = popDouble(stack);
         switch (operacao){
            case '*':
               resultado = op2*op1;
               break;
            case '/':
               resultado = op2/op1;
               break;
            case '+':
               resultado = op2+op1;
               break;
            case '-':
               resultado = op2-op1;
               break;
         }
         push(stack, &resultado, t_double);
      }
   }
   return popDouble(stack);
}
```

A função *calculate* recebe como argumento a fila posfixa e, enquanto existem elementos, ele realiza o seguinte procedimento: verifica o tipo de dado do primeiro elemento, caso seja um um double, ele remove o número e o insere na pilha, criada para calcular o resultado. Caso seja um caractere, ou seja, um operador, ele realiza o procedimento de cálculo: retira os dois últimos números da pilha e realiza a operação desejada na seguinte ordem, 2º número retirado <operador> 1º número retirado. Depois de processados todos os números e operadores da fila de entrada, é retornado o número armazenado na pilha de cálculo. Vale ressaltar que essa função pressupõe uma fila válida de entrada, ou seja, restará apenas um número na pilha de cálculo, portanto não verifica essa condição, que foi verificada na tradução. Essa função possui uma complexidade de ***O(n)***, devido ao laço *while* que percorre a fila de entrada ***n*** vezes, onde $\boldsymbol{n=n_{operadores}+n_{operandos}}$.

IV – Interface de Usuário

Para melhorar a experiência de uso da calculadora desenvolvida, foi criado uma interface gráfica utilizando o *port* para **Python 3** do framework **Qt**, uma vez que esta linguagem é eficiente, leve e sintética. Esse framework, desenvolvido em **C/C++** é formado por diversas bibliotecas para diferentes funcionalidades, em especial para desenvolvimento *GUI*, mas incluindo diversas como: redes TCP/IP, UDP, destaque de sintaxe, etc. Para interfacear o programa em **C** e os gráficos em **Python 3**, utilizamos a biblioteca ..., inclusa nos pacotes padrão da linguagem. O código da interface gráfica está disponível como executável e fonte nos arquivos do projeto, para evitar que este relatório se torne excessivamente extenso não transcreveremos o fonte aqui, mas incluimos abaixo uma captura de tela da interface.

3 Testes

Durante o desenvolvimento foram utilizados diversos testes unitários para avaliar o funcionamento das partes e garantir o funcionamento do todo. Será transcrito neste relatório 1(um) teste por biblioteca, a saber: *list, queue, stack, inputs, transform, calculate.* Além disso, transcreveremos um teste geral. O restante dos testes, variando de 2-4 para cada uma serão disponibilizados junto com os fontes.

List:

```
#include "../list.h"
#include <stdio.h>
#include <stdlib.h>

int main(){
    t_list *list = newList();
    double x=3,y=1,z=2;
    addStart(list, &x, t_double);
    addStart(list, &y, t_double);
```

```
    addStart(list, &z, t_double);
    double saindo = removeFirstDouble(list);
    while(saindo != 0){
        printf("%lf\n", saindo);
        saindo = removeFirstDouble(list);
    }
    return 0;
}
```

Queue:

```
#include "../queue.h"
#include <stdio.h>
#include <stdlib.h>

int main(){
    t_queue *queue = newQueue();
    double x=3,y=1,z=2;
    add(queue, &x, t_double);
    add(queue, &y, t_double);
    add(queue, &z, t_double);
    double saindo = removeDouble(queue);
    while(saindo != 0){
        printf("%lf\n", saindo);
        saindo = removeDouble(queue);
    }
    return 0;
}
```

Stack:

```
#include "../stack.h"
#include <stdio.h>
#include <stdlib.h>

int main(){
    t_stack *stack = newStack();
    double x=3,y=1,z=2;
    push(stack, &x, t_double);
    push(stack, &y, t_double);
    push(stack, &z, t_double);
    double saindo = popDouble(stack);
    while(saindo != 0){
        printf("%lf\n", saindo);
        saindo = popDouble(stack);
    }
    return 0;
}
```

Inputs:

```
#include "../inputs.h"
#include <stdio.h>

int main(){
	int i;
	char dado[] = "B+-*/() 0";

	for(i = 0; i < 9; ++i){
		printf("is %c a number? R: %d\n", dado[i],
isNumber(&dado[i]));
		printf("is %c an operator? R: %d\n", dado[i],
isOperator(&dado[i]));
		printf("is %c a parenthesis? R: %d\n",
dado[i], isParenthesis(&dado[i]));
	}
}
```

Transform:

```
#include "../transform.h"

void imprimir(t_queue * queue){
	int tipoDado;

	while(!isEmptyQueue(queue)){
		tipoDado = getNextQueueType(queue);
		switch(tipoDado){
			case t_double:
				printf("%.0f",
removeDouble(queue));
				break;
			case t_char:
				printf("%c",
removeChar(queue));
				break;
		}
	}
	printf("\nFila vazia\n");
}

int main(){
	t_queue * queue = newQueue();
	getExpression(queue);
	t_queue * saida = transform(queue);
	imprimir(saida);

	return 0;
}
```

Calculate:

```
#include "../queue.h"
#include "../calculate.h"

int main(){
    t_queue *queue = newQueue();
    double valor = 2;
    char op = '+';
    add(queue, &valor, t_double);
    valor = 3;
    add(queue, &valor, t_double);
    add(queue, &op, t_char);
    printf("fila gerada! 2 3 +\n");
    printf("%lf\n",calculate(queue));
    return 0;
}
```

4 Conclusão

O trabalho foi de grande utilidade para a compreensão e treinamento do uso de listas para implementar pilhas e filas. O conhecimento adquirido durante as aulas foi suficiente para implementar corretamente as funções de manipulação das listas. Houveram algumas dificuldades na implementação de funções de nível mais alto, como a de validação, devido ao número de casos que poderiam ocorrer. A escolha de implementar uma lista mista foi algo que apresentou também algumas dificuldades, em razão do cuidado que se deve ter ao lidar com ponteiros para *void.* Para facilitar o desenvolvimento de código compartilhado, utilizamos o GitHub, e o trabalho está disponível no repositório https://github.com/lucasaafaria/Trabalhos-ED/tree/master/Trabalho1 .

5 Bibliografia

- Tenenbaum, A. A; Langsan Y.; Augenstein M. Estruturas de Dados Usando C. Prentice-Hall, 1995.
- FARIAS, R. Estrutura de Dados e Algoritmos. **http://www.cos.ufrj.br**, 2009. Disponível em: < http://www.cos.ufrj.br/~rfarias/cos121/pilhas.html >. Acesso em: 1 abr. 2018.
- FARIAS, R. Estrutura de Dados e Algoritmos. **http://www.cos.ufrj.br**, 2009. Disponível em: < http://www.cos.ufrj.br/~rfarias/cos121/filas.html >. Acesso em: 1 abr. 2018.
- FARIAS, R. Estrutura de Dados e Algoritmos. **http://www.cos.ufrj.br**, 2009. Disponível em: < http://www.cos.ufrj.br/~rfarias/cos121/aula_11.html >. Acesso em: 1 abr. 2018.