Trabalho Prático 1

Gustavo Dias Apolinário - 2022035911

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

gustavo-apolinario@ufmg.br

1. Introdução

O problema proposto foi a implementação de um resolvedor automático de exercícios matemáticos (incluindo operações de soma, subtração, multiplicação e divisão), como bônus esse programa deveria também ler a expressão numérica e converter da notação usual (infixa) para a notação polonesa inversa (posfixa) ou no sentido contrário. É importante ressaltar que a expressão deve ser armazenada por meio de uma estrutura de dados adequada para o contexto. Dessa forma, a solução empregada foi representar internamente a entrada em formato de texto (string) em uma árvore binária.

2. Método

O programa foi desenvolvido na linguagem C++, a partir do compilador GCC da GNU Compiler Collection. Sistema Operacional: Linux Ubuntu 22.04.01 LTS.

2.1 Estruturas de Dados

O programa utilizou de duas estruturas de dados primárias: Árvore Binária e Pilha, duas estruturas secundárias: TipoCel e TipoNo. A árvore binária é uma estrutura organizada de forma hierárquica em que cada nó possui no máximo dois filhos (subárvores à esquerda e à direita), sendo que um nó é uma estrutura de dados que contém uma chave (valor) e duas referências para seus filhos (nós): um filho à esquerda e um filho à direita. Essa estrutura foi escolhida para armazenar a expressão, pois cada operação possui dos termos e um operador: a + b, mesmo que a ou b sejam compostos de outras operações é válido afirmar que toda operação só se preocupa com dois termos a e b. Essa natureza de 1 (operador) para 2 (números/termos) é congruente com a natureza da árvore binária de um nó para dois filhos. Além disso, ela se prova muito eficiente para o nosso problema, uma vez que é possível representar tanto a notação posfixa quanto infixa na mesma árvore, tendo em vista que a organização dela é voltada para as operações e ordem destas, característica que se mantém independente da notação. Por último, é útil o fato de

que as folhas (nós que não possuem filhos) são os números e os que não são folhas são sempre operadores.

Além disso, foi utilizada uma pilha, uma estrutura de dados em que temos várias células, compostas por um item (valor) e uma referência à célula anterior. O intuito da pilha é empilhar elementos como em uma estante em que se retira primeiro o último elemento que foi colocado, pense que se você faz uma pilha de livros e tenta retirar o primeiro que foi colocado, aquele que está mais abaixo, a pilha cai. A pilha era extremamente adequada para auxiliar no armazenamento da expressão posfixa, uma vez que se precisa armazenar os elementos na ordem correta na árvore e a pilha ajuda com essa ordem, pois na notação posfixa quando se encontra um operador, realiza-se a operação indicada nos últimos dois números lidos. Essa similaridade entre retirar o último elemento da pilha primeiro e realizar a operação no último elemento lido é fundamental para o funcionamento do programa.

2.2 Classes

O programa é composto de 4 classes, uma para cada estrutura de dados explicada no tópico anterior, seja esta primária ou secundária. A classe ArvoreBinaria armazena e resolve a expressão, a Pilha simplesmente empilha e desempilha itens, TipoNo é a representação em código do conceito, um item e dois ponteiras para outros nós e o TipoCel um item e um ponteiro para outra célula.

2.3 Funções

Árvore Binária:

```
4   class ArvoreBinaria
5   {
6   private:
7   TipoNo *raiz;
8   std::string tipo;
9   void ApagaRecursivo(TipoNo* p);
10  public:
11  ArvoreBinaria();
12  void ArvoreBinariaBasica(TipoNo* r, TipoNo* e, TipoNo* d);
13  TipoNo* Insere(std::string item,TipoNo*p);
14  TipoNo* InsereE(std::string item,TipoNo*p);
15  TipoNo* InsereD(std::string item,TipoNo*p);
16  long double Resolve(TipoNo*p);
17  TipoNo* Get_Raiz();
18  void Set_Tipo(std::string);
19  void Imprimir(TipoNo *p);
20  void Limpa();
21  ~ArvoreBinaria();
22  };
23  #endif
```

ApagaRecursivo: Libera a memória alocada recursivamente em um caminhamento pos-ordem: subárvore esquerda, direita e nó.

ArvoreBinaria: Construtor que faz a raiz receber NULL.

ArvoreBinariaBasica: Cria a estrutura simples de raiz, um filho a esquerda e um a direita, de acordo com os parâmetros.

TipoNo* Insere: Recebe o primeiro elemento e o coloca como raiz da árvore. Retorna a raiz.

TipoNo* InsereE: Recebe um nó e um item, cria um nó e esse novo nó vira filho à esquerda do novo passado como parâmetro. O novo nó tem como item o parâmetro item. Retorna o nó criado.

TipoNo* InsereD: Recebe um nó e um item, cria um nó e esse novo nó vira filho à direita do novo passado como parâmetro. O novo nó tem como item o parâmetro item. Retorna o nó criado.

long double Resolve: Recebe a raiz como parâmetro e em um caminhamento pos ordem verifica se o nó é um operador, se for o caso realiza a operação entre o filho da esquerda e o filho da direita, gera exceção quando tentativa de divisão por 0.

void Limpa: Faz a raiz receber NULL e chama o ApagaRecursivo().

void Imprimir: Recebe a raiz e recursivamente imprime a expressão in ordem (filho esquerda, no, filho direita) se o tipo for infixa e imprime pos ordem se for posfixa.

TipoNo* Get_Raiz: Retorna raiz

Set_Tipo: Altera o tipo de acordo com o parâmetro, gera erro se não for posfixa ou infixa.

~ArvoreBinaria(): Chama Limpa.

long double ConverteString_Num(std::string s): uma função que não é método de árvore, mas está definida no arquivo cpp, para auxiliar a função resolve. A função simplesmete converte de string para long double.

Pilha:

```
class Pilha{
private:
    TipoCel* topo;
    int tamanho;
public:
    Pilha();
    void Empilha(TipoNo* x);
    TipoNo* Desempilha();
    bool Vazia();
    void Limpa();
    ~Pilha();
    TipoNo* Get_Item_Topo();
};
#endif
```

Pilha: Construtor, topo recebe NULL e tamanho recebe 0

Empilha: Gera uma nova célula com item valendo x, a nova célula aponta pro atual topo (último item colocado antes do que está sendo empilhado no momento), o topo recebe a nova célula e o tamanho aumenta.

Desempilha: Verifica se a pilha está vazia e gera exceção. O topo recebe o anterior ao topo atual, tamanho diminui, deleta a memória do atual topo e retorna o item que estava no topo que foi deletado.

Vazia: Verifica se o tamanho é zero se for sim, retorna true, senão retorna false.

Limpa: Enquanto **Vazia** retorna true **Desempilha**.

~Pilha: Chama **Limpa.**

Get_Item_Topo: Retorno o item da célula do topo.

```
class TipoCel{
private:
    TipoNo* item;
    TipoCel * ant;
public:
   TipoCel(TipoNo* i);
   void Set_Ant(TipoCel *x);
   TipoCel* Get_Ant();
   TipoNo* Get_Item();
};
#endif
```

TipoCel: Construtor, o item da célula recebe i e o ponteiro para o anterior NULL.

Set_Ant: O ponteiro ant recebe o parâmetro.

Get_Ant: Retorna o ponteiro ant.

Get_Item: Retorna o item.

```
class TipoNo
{
public:
TipoNo();
void Set_Item(std::string item);
private:
std::string item;
TipoNo *esq;
TipoNo *dir;
friend class ArvoreBinaria;
};
#endif
```

TipoNo: Construtor, item recebe uma string vazia, esq e dir recebem NULL.

Set_Item: O item do nó recebe o parâmetro da função.

Funções main.cpp

bool isOperator (char str): Retorna true se o parâmetro for + , - ,* ou /, senão retorna false.

void armazenainfixa (std::string entrada, int t, TipoNo*p): Armazena recursivamente a entrada na árvore, em que t é um estado (0 =raiz, 1= esquerda, 2 =direita) e p é o último nó inserido ou a raiz se for primeira chamada.

void armazenaposfixa (std::string entrada): Armazena na árvore a entrada de forma não recursiva com o auxílio de uma pilha.

void LER(std::string TIPOEXP, std::string EXP): De acordo com o parâmetro TIPOEXP, que pode ser POSFIXA ou INFIXA, verifica se a expressão EXP

é válida, se for válida chama a função armazena correspondente, senão gera exceção.

void INFIXA(): Verifica se há expressão armazenada, se não houver gera exceção. Se houver pega muda o tipo da árvore para infixa e chama **Imprimir** da árvore.

void POSFIXA(): Verifica se há expressão armazenada, se não houver gera exceção. Se houver pega muda o tipo da árvore para posfixa e chama **Imprimir** da árvore.

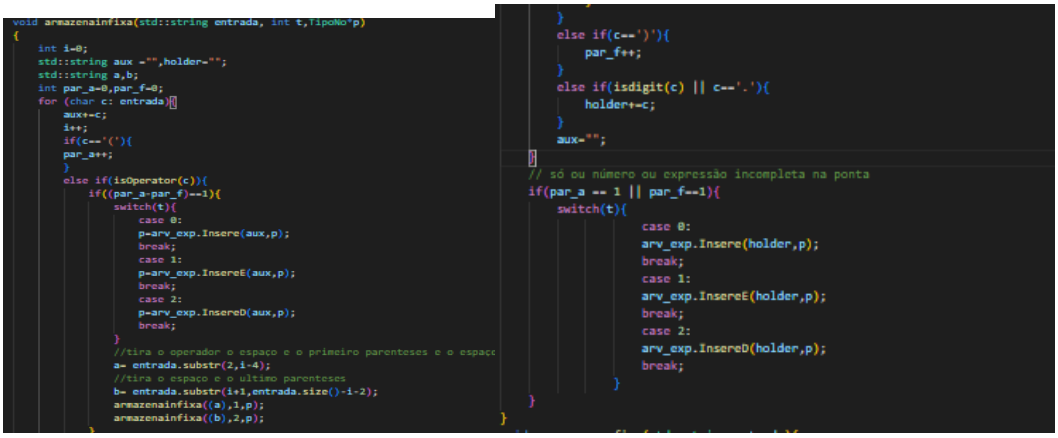
void RESOLVE: Verifica se há expressão armazenada, se não houver gera exceção. Se houver chama o **Resolver** da árvore.

3. Análise de Complexidade

```
void armazenaposfixa(std::string entrada){
    //Cria uma pilha de nós e garante que todos os nós cr
    Pilha p;
    TipoNo* no_p =NULL;
    TipoNo *aux1=NULL,*aux2=NULL;
    std::string holder="";
    for(int i =0; i<entrada.size();i++){
        if(isdigit(entrada[i])){
        while(entrada[i]!=' '){
        holder += entrada[i];
        i++;
        }
        no_p = new TipoNo();
        no_p->Set_Item(holder);
        p.Empilha(no_p);
        holder = "";
        }
        if(isOperator(entrada[i])){
            holder = entrada[i];
            aux2=p.Desempilha();
            aux1=p.Desempilha();
            no_p = new TipoNo();
            no_p->Set_Item(holder);
            holder="";
            arv_exp.ArvoreBinariaBasica(no_p,aux1,aux2);
            p.Empilha(no_p);
        }
}
}
```

A função armazenaposfixa lê a da esquerda para a direita e os números são inseridos na pilha, e quando um operador é encontrado, os dois números mais recentes são desempilhados, inseridos na árvore como filhos do operador encontrado e o operador é empilhado novamente como referência para as próximas inserções. Portanto, temos um for de tamanho n e dentro dele operações de custo O(1). Logo, a **complexidade de tempo** dessa função é **O(n).**

Complexidade de espaço: como a função cria um nó para cada número e cada operador teremos O (n – quantidade de espaços - quantidade de parênteses), logo O(n). As variáveis auxiliares são O(1), pois são criadas apenas uma vez. A pilha fica com tamanho máximo de n/2 (arredondado para o número inteiro superior), pois esse é a quantidade máxima de números que uma expressão posfixa pode ter, logo O(n). Portanto, O(n) + O(n) = **O(n) , complexidade de espaço**.



```
void armazenainfixa(std::string entrada, int t,TipoNo*p)
{
    int i=0;
    std::string aux ="",holder="";
    std::string a,b;
    int par_a=0,par_f=0;
    for (char c: entrada){
        aux+=c;
        i++;
        if(c=='('){
        par_a++;
        }
        else if(isOperator(c)){
            if((par_a-par_f)==1){
                switch(t){
                    case 0:
                    p=arv_exp.Insere(aux,p);
                    break;
                    case 1:
                    p=arv_exp.InsereE(aux,p);
                    break;
                    case 2:
                    p=arv_exp.InsereD(aux,p);
                    break;
                }
                //tira o operador o espaço e o primeiro parenteses e o espaço
                a= entrada.substr(2,i-4);
                //tira o espaço e o ultimo parenteses
                b= entrada.substr(i+1,entrada.size()-i-2);
                armazenainfixa((a),1,p);
                armazenainfixa((b),2,p);
            }
        else if(c==')'){
            par_f++;
        }
        else if(isdigit(c) || c=='.'){
            holder+=c;
        }
        aux="";
    }
    // só ou número ou expressão incompleta na ponta
    if(par_a == 1 || par_f==1){
        switch(t){
                    case 0:
                    arv_exp.Insere(holder,p);
                    break;
                    case 1:
                    arv_exp.InsereE(holder,p);
                    break;
                    case 2:
                    arv_exp.InsereD(holder,p);
                    break;
                }
    }
}
```

A função armazenainfixa possui um for com o tamanho da entrada e procurando o operador com maior prioridade, depois de encontrar insere o operador na árvore seja como raiz, filho à esquerda ou filho à direita dependendo da sua origem. Após armazenar retira o operador da expressão e separa o lado esquerdo da expressão do lado direito, tendo como o “meio” a posição do operador armazenado. A expressão da esquerda recebe o mesmo tratamento recursivamente com o parâmetro de inserir a esquerda e o equivalente ocorre à direita. Ao fim caso o parêntese seja único em um dos lados indica que a subárvore é apenas o número, nesse caso o número é inserido. Dessa forma, se insere apenas números e operadores. Se pensarmos por nível de chamada, a primeira chamada lê uma parte de n é interrompida chama duas funções recursivas e no retorno lê o resto de n, portanto ela lê n, uma fração da leitura antes e outra depois das chamadas recursivas, o nível 1. O segundo nível de chamada é composto pela chamada 1 da executada pelo nível anterior e a chamada 2 do nível anterior a chamada 1 vai ler só a parte esquerda de n (no mesmo molde que a anterior lê uma fração da parte esquerda de n antes das chamadas recursivas e lê uma fração depois, mas lê toda a parte esquerda de n) enquanto a chamada direita faz analogamente a parte da direita de n, ou seja o nível dois como um todo lê n. Supondo que o nível 3 finalmente alcançamos os números do lado direito, mas do lado esquerdo ainda há operações, leremos novamente n no nível 3. Porém, a partir do nível 4 lerá somente uma fração de n, pois no nível 3 já alcançamos os números se encerrando a recursividade em uma fração de n, dessa forma podemos considerar que por limite superior se lê n, pois n é superior a uma fração de n. Logo, podemos afirmar que as duas chamadas de recursão formam uma arvore binaria de chamadas e que o custo de espaço é equivalente à O(n*(nível da árvore de chamadas+1)) o nível da árvore de chamadas está relacionado aos operadores e a ordem deles, portanto em função de n temos O(n*constante da árvore). Essa constante é no máximo igual ao número de operadores + 1, uma vez que podemos considerar que cada operador gera um nível na árvore e ela já possui um nível inicial. Portanto, O(n*num_oper). O num_oper não cresce junto a n, pois podemos ter uma expressão infinitamente grande com dois números e apenas um operador. Dessa forma, o mais adequado na teoria seria dizer que a função é **O(n*num_oper) ou O(n^2)**, pois num_oper sempre menor ou igual a n. Contudo, na prática é esperado **O(n), pois o num_oper não irá crescer de forma relevante.**

Para tornar mais claro A árvore de chamadas não é a árvore que armazena os números é uma outra árvore secundária imaginária. Pense que a entrada inicial seja: (((5) + (4)) + (5)).

Nivel 0:	(((5) + (4)) + (5))		O(n)
Nivel 1:	((5) + (4))	(5)	O(n)
Nível 2:	(5)	(4)	inferior a O(n)

Portanto, podemos afirmar que teve custo O(n * 3(operadores + 1).

Complexidade de espaço: como a função cria um nó para cada número e cada operador teremos O(n – quantidade de espaços - quantidade de parênteses),

logo O(n). As variáveis auxiliares como “holder”, “i", “aux” são O(1), porém são criadas para cada chamada recursiva, se tornando O(n). Além disso, temos a pilha de execução das chamadas recursivas que também é O(n). Ao fim temos, O(n) + O(n) + O(n) = **O(n)**.

Função Ler:

Infixa: A complexidade de tempo de ler infixa é O(n),n igual ao tamanho da string de parâmetro, pois basicamente ela pega cada caractere da string e faz comparações de tamanho fixo, o número de comparações varia dependendo de qual for o caractere, porém o pior caso é 7 quando é um operador, além disso há comparações fixas no começo e ao final fora do loop que lê a string, somando 6. Portanto seria O(7*n + 6), logo O(n). Além disso, caso a expressão seja válida limpa a árvore que é uma operação O(n). Finalizando O(n) + O(n) = O(n).

Posfixa: Analogamente a complexidade de tempo é O(n), pois também pega cada caractere da string e faz comparações de tamanho fixo dependendo do tipo do caractere, sendo o número o pior caso com 4 comparações e 4 comparações fora do loop. Essa função também limpa a árvore se tudo estiver certo. Finalizando O(n) + O(n) =O(n).

Complexidade de espaço **infixa e posfixa:** Ambas funções possuem complexidade de espaço igual a O(h), sendo h a altura da árvore. Olhando exclusivamente para as funções a memória usada é constante e independente da entrada. Todas as variáveis são declaradas uma única vez fora de qualquer loop que envolva o tamanho da entrada. Basicamente temos alguns bools para verificar se certas condições foram atendidas, ponteiros e contadores, que são criados independente da entrada. Dessa forma deveria ser O(1). Entretanto, temos a chamada recursiva da limpeza da árvore, essa ocupa um espaço equivalente a altura da árvore em chamadas, pois a maior distância de chamadas será da raiz até a folha que é proporcional à altura.

Vale ressaltar que ao fim o ler infixa chama armazena infixa e suas complexidades são superadas pelo armazenainfixa e o mesmo ocorre com ler posfixa. Portanto, são mantidas as complexidade de aramzenainfixa para o ler infixa e o aramazenaposfixa para o ler posfixa. Apenas achei relevante a análise separada do que ocorria na leitura, mas a complexidade final está em aramazena.

Função INFIXA: Essa função tem **complexidade de tempo O(n)**, pois basicamente ela chama a função imprimir que percorre cada nó da árvore e executa uma operação constante de impressão para cada um dos n nós, logo O(n). Já a **complexidade de espaço é O(h)**, pois a função imprimir é recursiva e assim como explicado na função anterior a maior distância de chamadas será da raiz até a folha que é proporcional à altura.

Função POSFIXA: Exatamente a mesma ideia da **INFIXA** . Evidentemente, a infixa imprime em in-ordem e a posfixa em pos-ordem e a infixa tem os parênteses, entretanto o custo é o mesmo, pois segue a mesma ideia. **Complexidade de tempo**

O(n), executa uma operação constante de impressão a cada nó. **Complexidade de espaço O(h)**, pilha de chamadas recursivas proporcional à altura da árvore.

Função RESOLVE: A função tem **complexidade de tempo O(n)** e de **espaço O(h)**. Cada nó é visitado uma vez pela função recursiva resolve e o número de operações realizadas a cada nó e constante, pois as únicas operações são de fato a multiplicação, adição, subtração e divisão dependendo do caso, sendo a execução constante para cada nó e são n nós. Já a complexidade de espaço segue a mesma ideia da pilha de execução do tamanho da altura.

4. Estratégias de Robustez

Para tornar o programa robusto foram levadas em consideração diversas condições e caso fosse necessário era lançada uma exceção por meio de um “throw” e um conjunto try-catch exibia a mensagem que indica que houve um erro e indica qual foi esse erro.

- Expressão maior que mil caracteres é inválida
- Primeiro elemento infixa deve ser parênteses
- Caractere intruso em número decimal, exemplo: ‘+’ é intruso
- Número decimal deve ter apenas um ponto para indicar fração
- Dois números em sequência na infixa é inválido
- Primeiro elemento fora parênteses e espaço deve ser um número na infixa
- Operador imediatamente após abertura de parênteses não é válido (4 (+ 5)), devido à natureza dos parênteses
- Dois operadores em sequência também é inválido
- Fechamento de parênteses sempre após número ou parêntese fechado, não pode por operador ou abrir um parêntese e fechar sem conteúdo
- Último elemento deve ser número na infixa
- Número de parênteses abertos iguais ao de fechados
- No mínimo dois números e um operador para infixa e posfixa
- Número de ((operadores * 2) + 1) deve ser igual ao número de parênteses abertos, pois o primeiro operador gera três parênteses abertos (dois de número e um da expressão), caso realize uma operação com a expressão inicial com outra expressão são dois operadores e 4 parênteses abertos e se for expressão com número sozinho são dois parênteses abertos para um operador
- Caractere inválido, exemplo : ‘@’
- Último elemento da posfixa deve ser operador
- A diferença entre a quantidade de números e operadores na posfixa deve ser 1, pois cada par de números precisa de um operador e cada operação gera um número de forma que começa com dois números e um operador e se resulta em mais um número. Esse número ou é o resultado ou precisa de mais um número e mais um operador, somando 1 a cada prevalece a diferença inicial de dois números e um operador.
- **Verificação se expressão existe na conversão infixa**
- **Verificação se expressão existe na conversão posfixa**
- **Verificação se expressão existe no resolve**

- **Geração de erro quando tentativa de divisão por 0**
- Verifica se o comando é válido (como LER, RESOLVE, INFIXA, POSFIXA)
- Verifica se o arquivo foi aberto com sucesso
- Verifica na função desempilha se a pilha está vazia
- Únicos tipos possíveis “INFIXA” e “POSFIXA”
- Verifica se os parâmetros são válidos

5. Análise Experimental

Inicialmente era preciso analisar o programa em relação a sua funcionalidade e cumprimento da resolução do problema como um todo. Para isso, foram utilizados 32 exemplos de entradas, completamente distantas tantas de posfixas quanto infixas para testar se tudo era executado conforme o esperado.

Uma vez que a funcionalidade foi garantidade era preciso tentar quebrar o programa e ver o quão robusto ele era. Para isso, foram criados 24 exemplos, cada um com um erro tentando quebrar o programa, porém todos falharam.

A partir desse ponto era importante assegurar dois pontos: uso de memória correto e desempenho. Usando o Valgrind:

```
==6846== Memcheck, a memory error detector
==6846== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6846== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6846== Command: ./bin/programa bin/entdouble.s1.n5.i.in
==6846==
EXPRESSAO OK: ( ( ( ( 9.904341 ) + ( ( 5.733451 ) - ( 0.641665 ) ) ) - ( ( 2.165881 ) + ( 1.404730 ) ) ) - ( ( 5.732986 ) + ( ( 5.938726 ) - ( 8.993233 ) ) ) )
POSFIXA: 9.904341 5.733451 0.641665 - + 2.165881 1.404730 + - 5.732986 5.938726 8.993233 - + -
VAL : 8.74704


==6846==
==6846== HEAP SUMMARY:
==6846==     in use at exit: 0 bytes in 0 blocks
==6846==   total heap usage: 55 allocs, 55 frees, 85,385 bytes allocated
==6846==
==6846== All heap blocks were freed -- no leaks are possible
==6846==
==6846== For lists of detected and suppressed errors, rerun with: -s
==6846== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

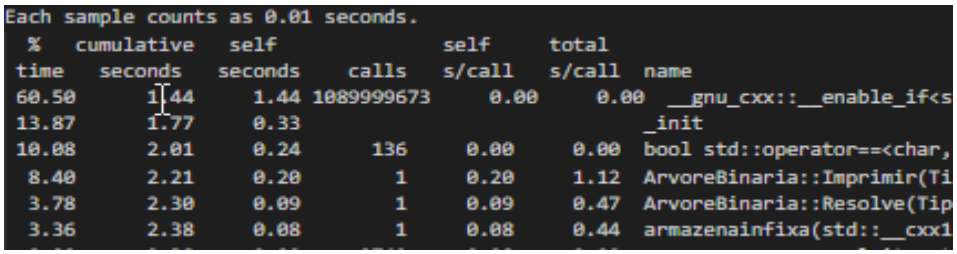
Podemos perceber pela mensagem que todos os blocos no heap foram liberados, o mesmo procedimento foi feito para um caso de posfixa e houve a mesma mensagem como reposta.

Por fim o desempenho de todas as funções seria muito extenso então irei colocar de três exemplos com infixa, pois houve uma peculiaridade em relação ao número de operadores.

Primeiro caso, 145 caracteres de entrada com 7 operadores:

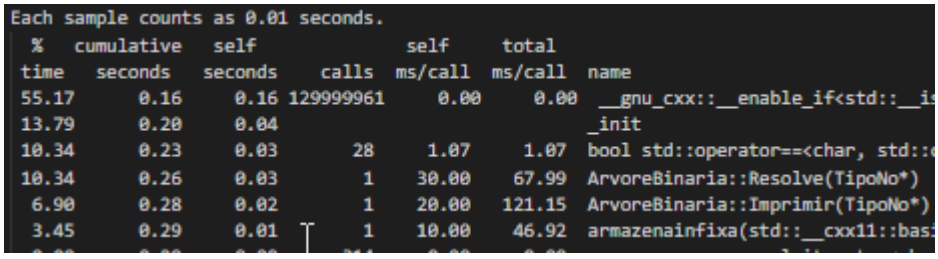
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 58.12      0.68     0.68 609999817     0.00     0.00  __gnu_cxx::__enable_if<std::__is_
 11.11      0.81     0.13        1   130.00   562.09  ArvoreBinaria::Imprimir(TipoNo*)
 10.26      0.93     0.12       86     1.40     1.40  bool std::operator==<char, std::c
  8.55      1.03     0.10                             _init
  7.69      1.12     0.09        1    90.00   272.56  ArvoreBinaria::Resolve(TipoNo*)
  4.27      1.17     0.05        1    50.00   217.21  armazenainfixa(std::__cxx11::basi
```

Primeiro caso, 179 caracteres de entrada com 13 operadores:



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 60.50      1.44     1.44 1089999673     0.00     0.00  __gnu_cxx::__enable_if<s
 13.87      1.77     0.33                             _init
 10.08      2.01     0.24      136     0.00     0.00  bool std::operator==<char,
  8.40      2.21     0.20        1     0.20     1.12  ArvoreBinaria::Imprimir(Ti
  3.78      2.30     0.09        1     0.09     0.47  ArvoreBinaria::Resolve(Tip
  3.36      2.38     0.08        1     0.08     0.44  armazenainfixa(std::__cxx1
```

Terceiro caso, 100 caracteres de entrada com 1 operador:



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 55.17      0.16     0.16 129999961     0.00     0.00  __gnu_cxx::__enable_if<std::__i
 13.79      0.20     0.04                             _init
 10.34      0.23     0.03       28     1.07     1.07  bool std::operator==<char, std::
 10.34      0.26     0.03        1    30.00    67.99  ArvoreBinaria::Resolve(TipoNo*)
  6.90      0.28     0.02        1    20.00   121.15  ArvoreBinaria::Imprimir(TipoNo*)
  3.45      0.29     0.01        1    10.00    46.92  armazenainfixa(std::__cxx11::bas
```

Fica claro que a função armazena infixa apesar da entrada, tem uma dependência com o número de operadores, tão relevante quanto o tamanho da entrada. As funções Resolve e Imprimir (Equivalente a Conversão) são O (n), sendo n os nós e os nós aumentam de acordo com o número de elementos, ou seja, ter apenas 2 números e um operador, contra uma entrada com muitos operadores e consequentemente muitos números, gera mais nós. Portanto, essa relação entre o tempo delas em cada caso é o esperado.

6. Conclusões

É possível concluir que o programa foi capaz de solucionar o problema proposto por completo, possuindo todas as funções de leitura, conversão e resolução dentro do formato esperado. As funções de cada estrutura de dados estão bem implementadas e cada uma realizando a função é responsável de forma simples e condizente com o conceito de cada estrutura. O cálculo da complexidade foi muito bem detalhado e baseado em provas empíricas, evidenciando que o a eficiência do programa estava muito boa e de acordo com o esperado para cada estrutura de dados. Além disso, o código está extremamente robusto com proteção contra mais de 30 condições inadequadas.

7. Bibliografia

- Geeksforgeeks.or

- Praharsh Bhatt : https://medium.com/

- https://www.techiedelight.com/

- https://stackoverflow.com/ https://en.wikipedia.org/wiki/Binary_tree

8. Instruções Para Compilação e Execução

- Abrir o terminal
- Acessar a pasta TP
  Comando: cd / <caminho para a pasta /TP >
- Compilar com o Makefile
  Comando: make
- Executar os arquivos com o Makefile
  Comando: make run
- Finalizado, caso deseja limpar:
  Comando: make clean

Vale ressaltar que todos os testes já estão dentro da pasta bin, não é possível com o formato atual dar make run em um arquivo fora da bin, caso queira adicionar um novo arquivo a ser testado é necessário adicioná-lo a bin do projeto e no makefile acrescentar: “./$(EXECUTABLE) $(BINDIR)/<nome_arquivo>” .