2 – *Stack* (Pilha)

2.0.1 - Introdução

A estrutura de dados pilha é uma estrutura abstrata na qual temos ao menos duas operações básicas, a operação de *push* e *pop*. Essas operações correspondem a inserção e remoção de elementos respectivamente. A principal característica de uma pilha está no seu comportamento, no qual segue o modelo *LIFO* (*Last in - First Out*) que define que o último elemento a ser inserido deve ser o primeiro elemento a ser retirado.

2.0.2 - Explicação

Temos como a definição de pilha uma estrutura que siga o comportamento *LIFO* (*Last in - First Out*) no qual o último elemento a ser inserido deve ser o primeiro elemento a ser retirado. Esse comportamento é o que define uma pilha, a forma como ela vai se comportar quando suas principais funções forem usadas. A forma de implementação de uma pilha pode ser das mais simples até as mais complexas e pode-se utilizar tanto a alocação de memória estática-sequencial ou a dinâmica-encadeada. Dependendo da forma como essa estrutura é implementada, podemos ter algoritmos com diferentes complexidades embora o mais comum seja uma implementação otimizada com operações em tempo constante.

Podemos exemplificar um uso da estrutura de pilha no dia a dia quando estamos ainda dormindo e temos o objetivo de ir trabalhar. Durante o processo de alcance desse objetivo principal várias outras atividades vão surgindo e tomando lugar em nossa pilha de tarefas. Sendo assim podemos representar esse processo como uma pilha de tarefas na qual as tarefas de maior prioridade se encontram no topo da pilha:

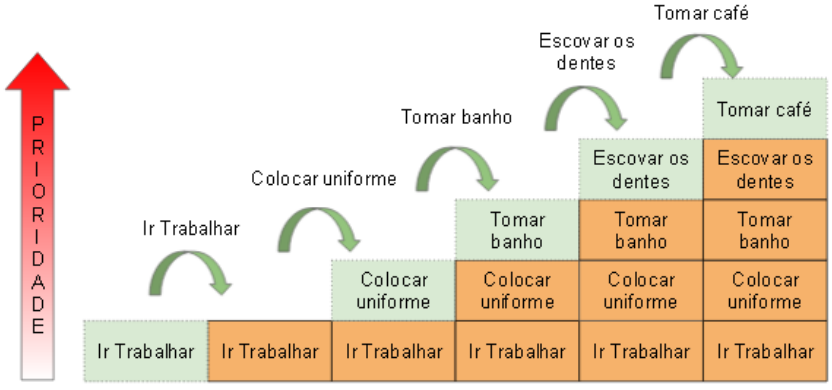

Imagem - Inserção de tarefas na pilha

Como podemos ver na figura ao lado, uma pessoa tem o objetivo de ir trabalhar e esse é o seu objetivo principal, no entanto, no meio do caminho surgem outras tarefas que precisam ser realizadas para que finalmente ela esteja pronta para ir trabalhar. Esse é um caso típico de utilização de uma estrutura de pilha.

O processo de inserir novos valores na pilha é chamado de ***push*** e mais a frente veremos que esse processo tem uma complexidade de O(1) ou constante. Para garantirmos que a nossa pilha cumpra o requisito de *LIFO* precisamos nos certificar de que a operação de remoção também funcione de forma correta. Seguindo o mesmo exemplo podemos ver agora como a nossa pilha se encontra após essa inserções:

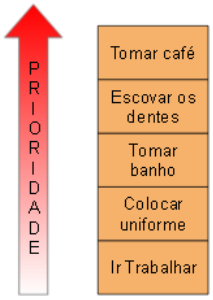

Imagem - Estado da pilha atual

A nossa estrutura de pilha se encontra agora com cinco elementos, e como sabemos, o primeiro elemento que foi inserido na pilha foi a tarefa “Ir Trabalhar” e o último a ser inserido foi “Tomar café”. Para que a nossa pilha cumpra com a sua função devemos implementar a função de remoção que é conhecida como ***pop*** de forma que o primeiro elemento a ser removido seja o último inserido, nesse caso, o “Tomar café” deve ser retirado primeiro. Assim como a operação *push, pop* também tem complexidade de O(1).

Vemos que para que a pilha cumpra o seu objetivo, a remoção dos valores deve ocorrer do mais recente para o mais antigo. Assim podemos ver na imagem ao lado que as tarefas com maior prioridade e as que se encontram no topo da pilha são as primeiras a serem executadas e retiradas da pilha.
Chegamos ao fim da pilha como a última remoção que é a tarefa original que tínhamos de executar, nesse exemplo: “Ir Trabalhar”.

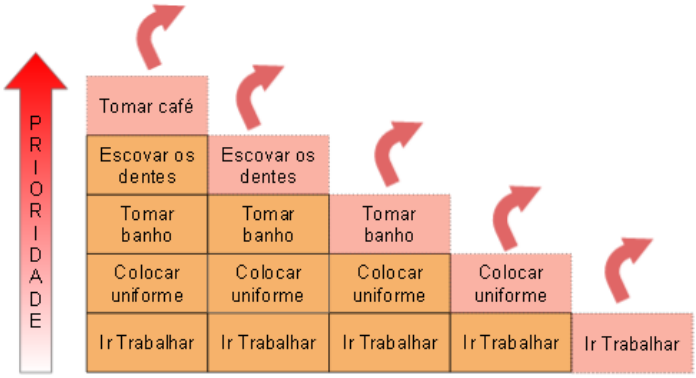

Imagem - Remoção de tarefas na pilha

É lógico considerar que uma das utilizações da pilha consiste em armazenar tarefas que são momentaneamente interrompidas para dar vez a outras tarefas que, de alguma forma, precisam ser executadas. Como no exemplo: para ir trabalhar a pessoa precisa antes colocar o uniforme que por sua vez deve ser realizado após tomar banho, que por sua vez ocorre após o processo de escovar dentes e esse, claro vem após tomar o café. Assim temos um exemplo prático do dia a dia que explica a utilização de uma estrutura do tipo *stack* ou pilha.

2.0.3 - Complexidade

Uma *stack* possui ao menos duas operações básicas e elas são: ***push*** e ***pop***, respectivamente, inserção e remoção. Essas operações são utilizadas a todo o momento sendo assim, o ideal é que tais operações sejam realizadas no menor tempo possível ou como a menor complexidade. Vamos analisar os processos de *push* e *pop* de maneira separada, analisar o seu funcionamento e complexidade.

Push

O processo de *push* consiste em adicionar valores a uma pilha. Cada implementação pode ser diferente uma das outras dependendo do tipo de pilha, se é de alocação de memória estática-sequencial ou dinâmica-encadeada. No entanto, em ambos os casos, conseguimos obter uma complexidade O(1) ou tempo constante. Consideremos uma pilha de lista sequencial de tamanho 5 que armazena valores inteiros.

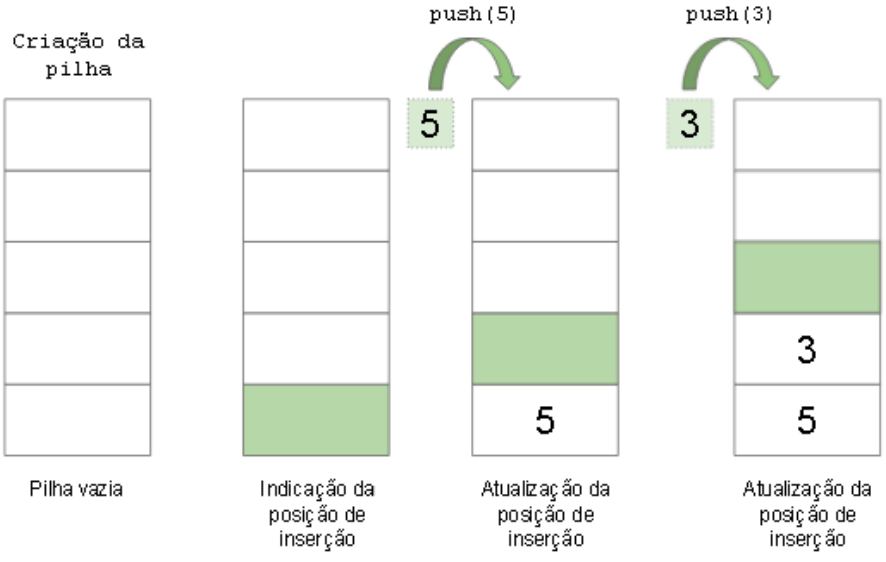

Imagem- Inserção de valores na pilha

Para poder inserir valores na pilha devemos saber onde inserir os novos valores e isso pode ser obtido por armazenar o local disponível para inserção, que na imagem acima está representado de verde. Sabendo onde o próximo valor será inserido, basta verificarmos se é possível inseri-lo e então realizar a operação de *push.*

Tendo em vista que já temos a posição da inserção e que basta colocar o valor na posição desejada, chegamos a conclusão que a complexidade dessa operação é **O(1)** ou constante. Todos os valores são inseridos no topo da pilha, assim não são necessários mais passos para executar essa operação. Se tivermos uma lista com 5, 500 ou N elementos a complexidade sempre será O(1).

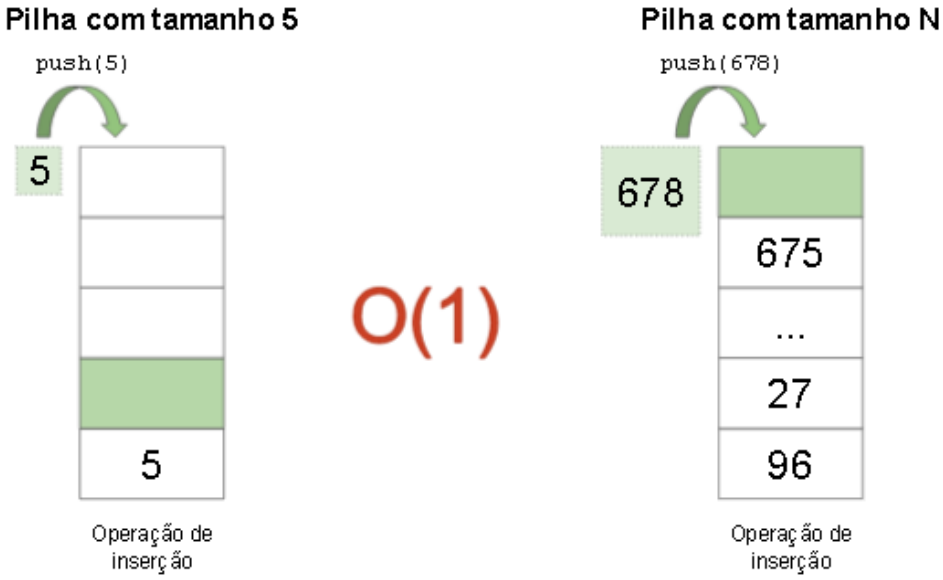

Imagem - Operação *push* em pilhas de diferentes tamanhos

*Vale a pena ressaltar que sim podemos ter pilhas que não são O(1) o que seria uma implementação bem ruim dessa estrutura, visto que é simples realizarmos o procedimento de *push* em tempo constante.

Pop

O processo de *pop* consiste em remover valores a uma pilha. Cada implementação pode ser diferente uma das outras dependendo do tipo de pilha, se é de alocação de memória estática-sequencial ou dinâmica-encadeada. No entanto, em ambos os casos conseguimos obter uma complexidade **O(1)** ou tempo constante. Consideremos uma pilha de lista sequencial de tamanho 5 que armazena valores inteiros.

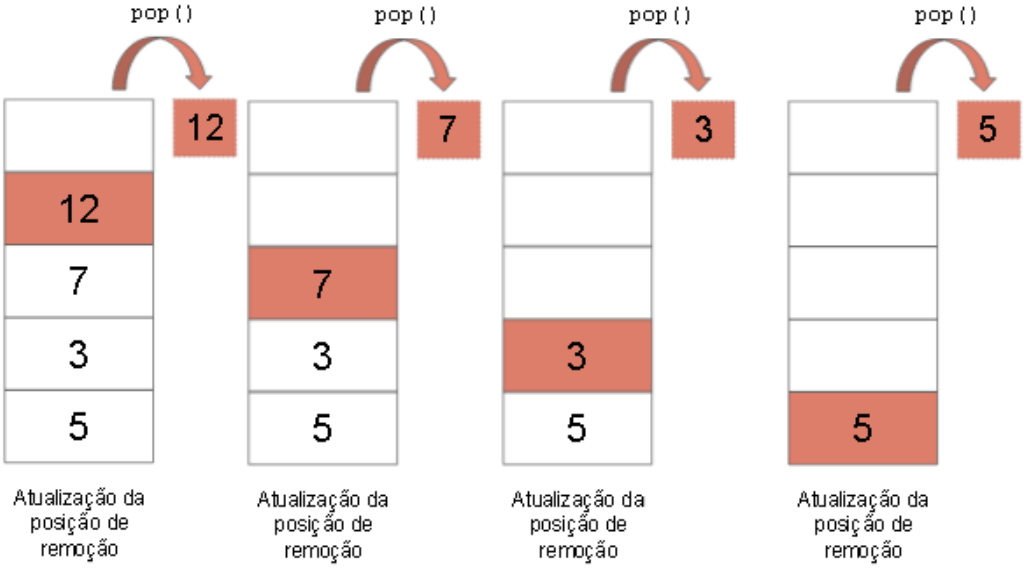

Imagem - Remoção de valores da pilha

Para poder remover valores na pilha devemos saber de onde os valores serão retirados e isso pode ser obtido por armazenar o local da próxima remoção, que na imagem acima está representado de vermelho. Sabendo de onde o próximo valor será removido, basta verificarmos se é possível removê-lo e então realizar a operação de *pop*.

Tendo em vista que já temos a posição da remoção e que basta remover o valor da posição desejada, chegamos a conclusão que a complexidade dessa operação é **O(1)** ou constante. Todos os valores são removidos do topo da pilha, assim não são necessários mais passos para executar essa operação. Se tivermos uma lista com 5 ou N elementos a complexidade sempre será O(1).

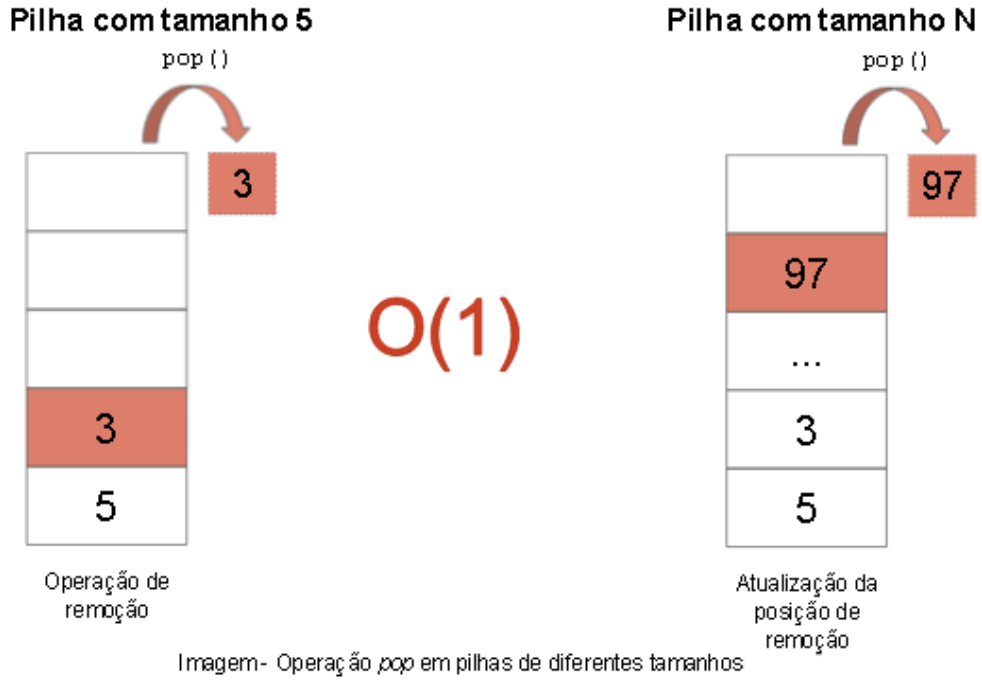

Imagem- Operação *pop* em pilhas de diferentes tamanhos

*Podemos ter pilhas que não são O(1) o que seria uma implementação bem ruim dessa estrutura visto que é simples realizarmos o procedimento de *pop* em tempo constante.

Operações

Considerando o que foi dito acima temos que tanto a operação de *push* quanto a operação de *pop* são **O(1)** e com essas característica temos uma implementação de pilha com complexidade ideal.

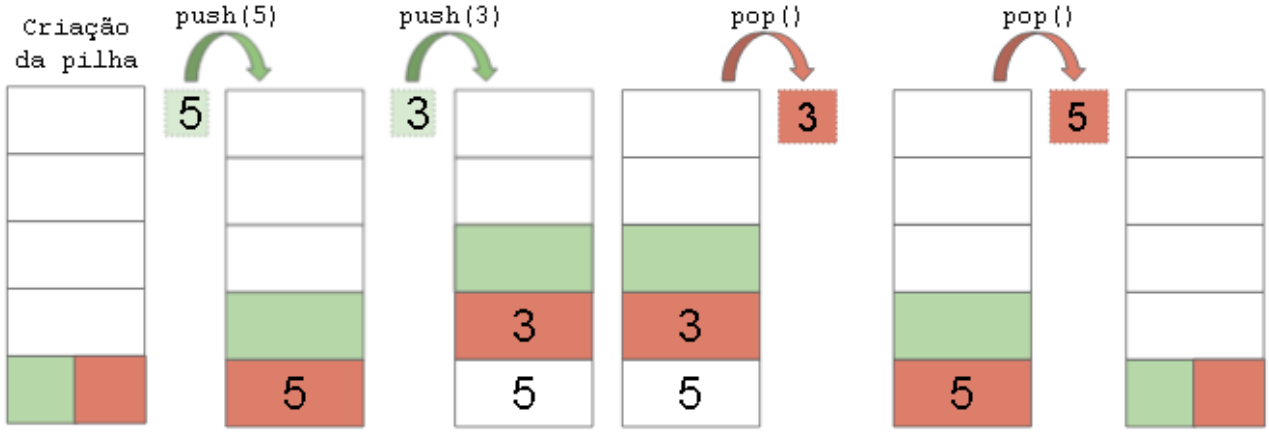

Imagem - Exemplo manipulação de uma pilha

2.0.4 – Implementação utilizando lista sequencial e uso.

```
#include <iostream>
using namespace std;
/*
    Implementação de uma pilha usando lista sequencial.
    Autor: Kevin R.
*/

#define TAM 5
typedef char ItemType;

struct Stack
{
    ItemType data[TAM];
    int length = 0 ;
};

bool isEmpty(Stack &p)
{
    return ( p.length == 0 );
}

bool isFull(Stack &p)
{
    return ( p.length == TAM );
}

void push(Stack &p, ItemType n)
{
    if (!isFull(p))
    {
        p.data[p.length] = n;
        p.length++;
    }
    else{
        cout <<  "Stack cheia!"<<endl;
    }
}
```

```
ItemType pop(Stack &p)
{
    if (!isEmpty(p))
    {
        int tmp = p.data[p.length-1];
        p.length--;
        return tmp;
    }
    else
    {
        cout <<  "Stack vazia!"<<endl;
        return false;
    }
}
/**
 * Função que exibe na tela uma representação da pilha.
 */
void describe(Stack p)
{
    if (isEmpty(p))
    {
        cout << "Stack: vazia" << endl;
        return;
    }

    cout << "Stack: ";
    for (int i = 0; i < p.length; i++)
    {
        cout << p.data[i] << " ";
    }
    cout << endl;
}
```

```
int main()
{
    //criação da pilha com o nome p
    Stack p;

    //exibe a pilha, que se encontra vazia.
    describe(p);


    /* O processo abaixo lê caracteres até que seja diferente de '\n' ou Enter
*/
    ItemType c;
    cin.get(c);

    while(c != '\n')
    {
        push(p,c);
        cin.get(c);
    }

    /* O processo abaixo desempilha os valores empilhados acima, e caso o número
    de caracteres empilhados seja maior que o tamanho da pilha será exibida uma
mensagem
    informando que a pilha está cheia. */

    while(!isEmpty(p))
    {
        cout << pop(p);;
    }

    return 0;
}
```

O código acima contém a implementação da *stack* ou pilha usando uma lista sequencial. Caso queira modificar o tamanho da lista altere `#define TAM 5` no lugar do 5 coloque o tamanho que deseja.
A pilha possui as duas operações principais: `push` e `pop` e também outras operações como: `isEmpty`, `isFull e describe`. Que respectivamente , retorna se a pilha está vazia, retorna se a pilha está cheia e exibe o conteúdo da pilha.

Entrada	Saída
kevin	nivek
54321	12345
abcde	edcba

O programa da função `main` faz a criação de uma pilha, e lê vários caracteres até que algum seja '\n' ou o usuário dê Enter. E logo depois os valores inseridos são desempilhados e exibidos na tela.

2.0.5 – Implementação usando lista encadeada e uso.

```
#include <iostream>
using namespace std;

typedef char ItemType;

struct Node{
    ItemType v;
    Node* next = NULL;
};

struct Stack{
    Node* node = NULL;
};


bool isEmpty(Stack *p)
{
    return (p->node == NULL);
}

bool isFull(Stack *p)
{
    try
    {
        Node *newnode = new Node;
        delete newnode;
        return false;
    }
    catch(std::bad_alloc e)
    {
        return true;
    }

}

void push(Stack* &p, ItemType n)
{
    if(!isFull(p))
```

```
    {
        Node *newnode = new Node;

        newnode->v = n;
        newnode->next = p->node;

        p->node = newnode;
    }
    else
    {
        cout << "Stack cheia! " << endl;
    }


}

ItemType pop(Stack* &p)
{
    if(!isEmpty(p))
    {

        Node *tmpnode  = p->node;
        ItemType tmp = tmpnode->v;
        p->node = p->node->next;
        delete tmpnode;

        return tmp;
    }
    else
    {
        cout << "Stack vazia!" << endl;
        ItemType n;
        return n;
    }

}

/**
 * Função que exibe na tela uma representação da pilha.
 */
```

```
void describe(Stack *p)
{
    Node *aux = p->node;

    if (aux == NULL)
    {
        cout << "Stack Vazia" << endl;
        delete aux;
    }else
    {
        cout << "Stack: " ;
        for(Node *aux2 = p->node; aux2 != NULL; aux2 = aux2->next)
        {
            cout << aux2->v << " ";
        }
        cout << endl;
    }
}
int main()
{

    //criação da pilha com o nome stack
    Stack *stack = new Stack;

    /* O processo abaixo lê caracteres até que seja diferente de '\n' ou Enter
*/
    ItemType c;
    cin.get(c);

    while(c != '\n')
    {
        push(stack,c);
        cin.get(c);
    }

    /* O processo abaixo desempilha os valores empilhados acima, e caso o número
    de caracteres empilhados seja maior que o tamanho da pilha será exibida uma
mensagem
    informando que a pilha está cheia. */
    while(!isEmpty(stack))
```

```
    {
        cout << pop(stack);
    }

    return 0;
}
```

O código acima contém a implementação da *stack* ou pilha usando uma lista encadeada. A pilha possui as duas operações principais: `push` e `pop` e também outras operações como: `isEmpty, isFull e describe`. Que respectivamente , retorna se a pilha está vazia, retorna se a pilha está cheia e exibe o conteúdo da pilha.

Entrada	Saída
ed1listaligada	adagilatsil1de
kevinrodrigues	seugirdornivek
computacaonauff	ffuanoacatupmoc

O programa da função `main` faz a criação de uma pilha, e lê vários caracteres até que algum seja '\n' ou o usuário dê Enter. E logo depois os valores inseridos são desempilhados e exibidos na tela.

2.0.6 – Comparação entre pilha com lista encadeada e sequencial

Pilha		
Critério	**Lista sequencial**	**Lista encadeada**
BigO push	O(1)	O(1)
BigO pop	O(1)	O(1)
Uso de memória	Fixo	Variável
Dificuldade de implementação	Fácil	Fácil
Custo de memória por dado	Custo do dado	Custo do dado + custo do ponteiro

Algumas coisas precisam ser pontuadas quanto a utilização das pilhas em diferentes implementações. Quando usamos pilhas de lista sequencial é necessário definir o tamanho da nossa pilha, ou o tamanho máximo de valores que ela poderá armazenar. No código da seção 2.0.4 isso pode ser feito na linha: `#define TAM 5` no lugar do 5 coloque o tamanho que deseja que a sua lista tenha. Assim você terá certeza de que sua lista suportará aquela capacidade de itens durante o seu uso.

Já na pilha de lista encadeada você não precisa informar e alocar previamente os espaços necessários para uso. O início da pilha é na verdade um ponteiro para o primeiro elemento de uma lista encadeada.Todo o processo é realizado dinamicamente, no entanto pode ocorrer que em algum momento o seu programa fique sem memória disponível e isso pode fazer com que a sua pilha não consiga mais inserir valores. No código da seção 2.0.5 temo o seguinte trecho de código da função `isFull`:

```
bool isFull(Node *p)
{
    try
    {
        Node *newnode = new Node;
        delete newnode;
        return false;
    }
    catch(std::bad_alloc e)
    {
        return true;
    }
}
```

Essa função usa um trecho de código protegido com o comando *try catch*, e ele foi usado porque pode ser que na hora de alocar mais memória `Node *newnode = new Node;` não seja possível pois não há mais memória disponível para ser alocada. Assim o código gerará um erro do tipo `bad_alloc` indicando que houve erro ao alocar memória e com isso podemos receber esse erro e tratá-lo sem que o nosso programa pare de funcionar. No caso da função `isFull` caso recebamos um erro do tipo *bad_alloc* sabemos que não há mais memória disponível para alocação e o retorno da função é `true` indicando que sim, a pilha está cheia, visto que não podemos mais inserir valores nela.

Em termos de complexidade, ambas são equivalentes, visto que tanto na pilha com lista sequencial quanto na pilha de lista encadeada a complexidade das operações é de O(1).

Considerando o que vimos, temos que, em determinados casos, será interessante usarmos pilhas com alocações estáticas, principalmente quando sabemos o máximo de valores e queremos garantir a utilização da quantidade máxima desejada. De qualquer forma não há uma regra, a tomada de decisões quanto a utilização de uma ou de outra implementação da pilha fica a cargo do desenvolvedor.

2.1.0 – Pilha e Calculadora Polonesa Reversa

Uma das utilizações comuns para estrutura de pilha é a aplicação de suas propriedades para criação de uma calculadora polonesa reversa. Essa calculadora funciona de forma ligeiramente diferente da qual estamos habituados. No qual inserimos primeiramente os operandos e logo depois as operações, justamente essa forma de organizarmos os operandos e operadores é que lhe dá o seu nome como calculadora polonesa reversa. Para entendermos bem o funcionamento do projeto como um todo, vamos dividi-lo em partes. Todos os conceitos para a compreensão da pilha podem ser encontrados no capítulo 2, das seções 2.0.1 a 2.0.6.

2.1.1 – Notação Polonesa Reversa

Geralmente ao resolvermos uma operação matemática seguimos o padrão `x + y`, onde `x` e `y` são os operandos e `+` é a operação que desejamos realizar. No entanto, quando queremos realizar algumas operações em que a prioridade comum não pode ser representada diretamente, acabamos por utilizar os parênteses para forçarmos uma determinada operação. Como no caso de fazer primeiro uma soma e depois pegar esse resultado e multiplicar por um outro valor:

x	y	z	Exemplo I	x + y * z	2 + 3 * 5	2 + 15	17
2	3	5	Exemplo II	(x + y) * z	(2 + 3) * 5	5 * 5	25

No exemplo `I` realizamos primeira a multiplicação e logo depois a soma. Nesse caso as ordens das operações estão definidas por regras de prioridades e quando queremos forçar uma operação devemos fazer uso dos parênteses como mostrado no exemplo `II`. Esse tipo de notação que geralmente usamos para realizarmos operações aritméticas é conhecido como notação infixa. No entanto, há uma forma de representar a mesma operação realizada no exemplo `II` sem a utilização dos parênteses, e uma das formas de fazermos isso é por usar a notação posfixa ou notação polonesa reversa.

Diferentemente da notação infixa, a notação polonesa reversa recebe primeiramente os seus operandos e depois os operadores. Assim, `x + y` seria representado `x y +`.

Notação infixa	Notação polonesa reversa
x + y * z	x y z * +
x * (y + z)	x y z + *
(x + y) / (z - l)	x y + z l - /
(x + y) / (z - l) * k	x y + z l - / k *

2.1.2 – Arquitetura da Calculadora Polonesa Reversa

Veremos nessa seção como podemos compreender a arquitetura de uma simples calculadora polonesa reversa. Basicamente para esse projeto precisaremos de uma pilha para armazenarmos os operadores e de uma unidade de processamento que será responsável por realizar as operações.

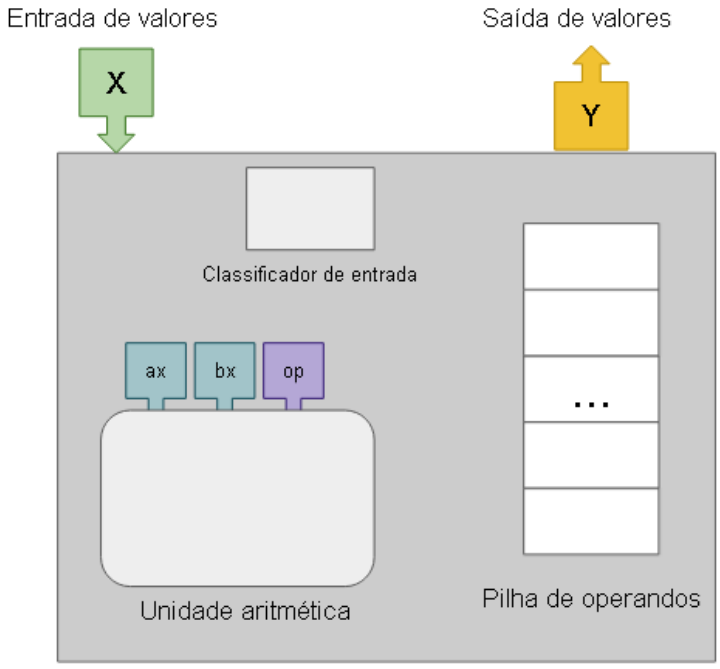

Imagem - Arquitetura simplificada da calculadora

Faremos uso de uma *pilha* para armazenarmos todos as variáveis ou operandos, precisaremos também de uma *unidade aritmética* que processará todos os valores se comunicando com a pilha para receber e enviar valores. Além dessas duas partes principais, teremos também variáveis temporárias como `ax`, `bx` e `op` que servirão para armazenar operandos e operadores. Todos os valores entrarão na nossa calculadora através de uma porta de *entrada de valores* que enviará esse dado para o *classificador de entrada* classificar a entrada como um operador ou operando. Caso seja operador enviar para `op`, caso seja operando envia para a *pilha.*

Na imagem abaixo podemos ver outra representação da arquitetura de uma forma mais detalhada.

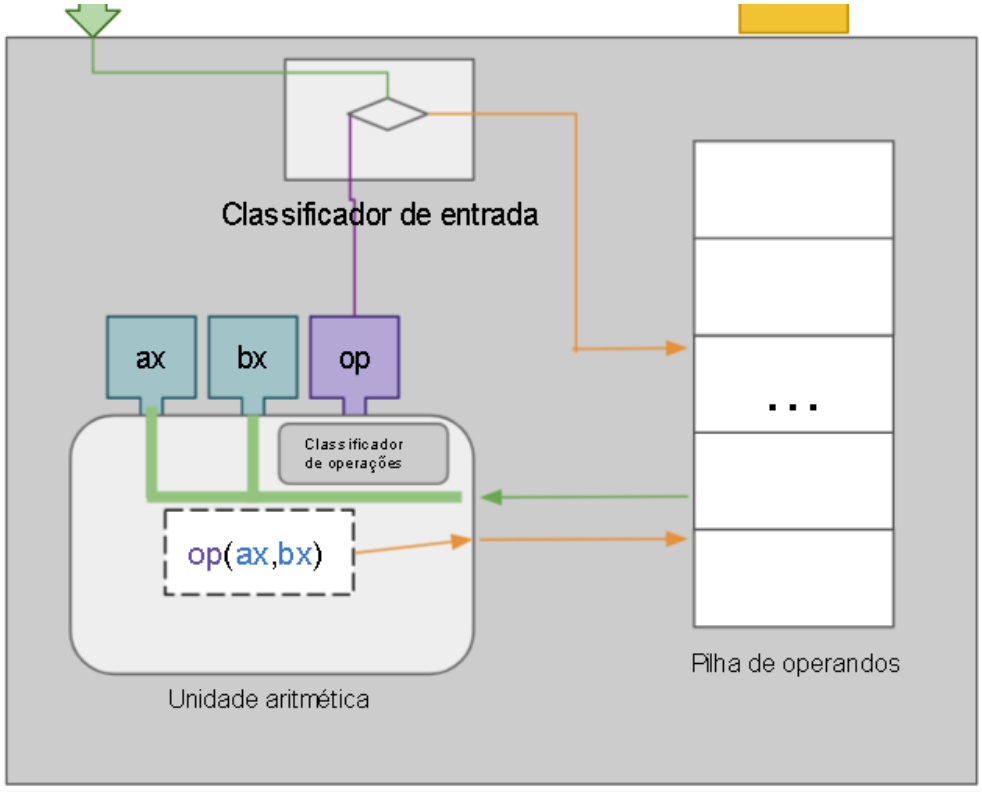

Imagem - Arquitetura da calculadora

2.1.3 – Pilha e Calculadora Polonesa Reversa

A essa altura você deve estar se perguntando como a estrutura de pilha se relaciona com a calculadora polonesa reversa. Bom vamos considerar o seguinte exemplo:

Notação infixa	Notação polonesa reversa	x	y	z
x * (y + z)	x y z + *	2	3	5

Para realizarmos a operação `x y z + *` devemos receber cada valor vez por vez, e quando encontrarmos um sinal de operação devemos operar os dois valores que serão retirados da pilha.

*Algumas verificações são necessárias para verificar se a operação é possível de ser realizada, por motivos didáticos estou considerando uma situação ideal.

Vejamos como podemos realizar a operação citada acima. Para isso vamos substituir `x y` e `z` por valores inteiros.

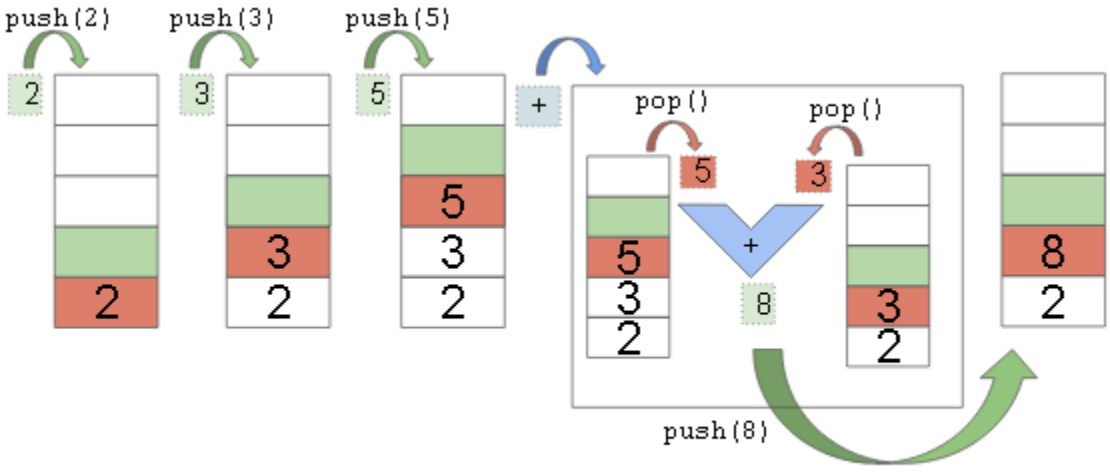

Imagem - Utilização de pilha na calculadora 01

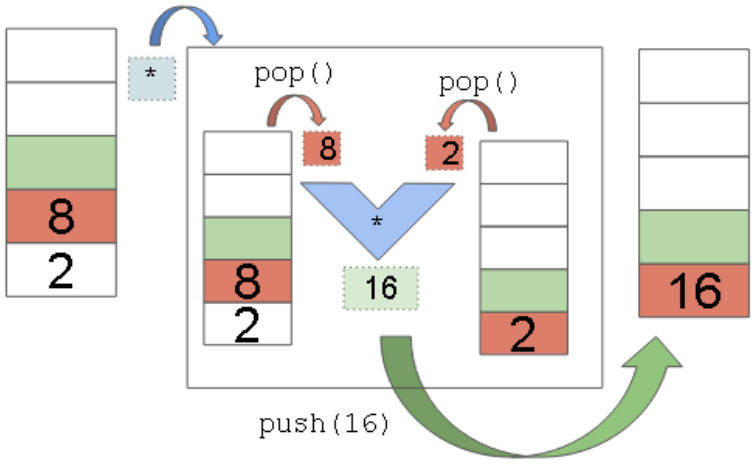

Imagem - Utilização de pilha na calculadora 02

2.1.3 – Implementação da Calculadora Polonesa Reversa com lista sequencial

```
#include <iostream>
#include <cmath>
using namespace std;

/*  Autor: Kevin Souza
    Calculadora Polonesa Inversa v4.0

    Uma calculadora feita para ser utilizada de verdade, por isso
    ela vem com vários recursos extras. Além de realizar operações matemáticas
    básicas ela possui uma pequena memória para armazenamento de valores.

*/

#define PI 3.14159265

/* Stack Size, max operands */
#define TAM 10

/* Tamanho máximo das memórias de numeros */
#define MEM_SIZE 10

/* Atalho de tipagem genérica */
typedef float ItemType;

struct Stack
{
    ItemType data[TAM];
    int length = 0;
};

bool isEmpty(Stack &p)
{
    return (p.length == 0);
}

bool isFull(Stack &p)
{
```

```
    return (p.length == TAM);
}

void push(Stack &p, ItemType n)
{
    if (!isFull(p))
    {
        p.data[p.length] = n;
        p.length++;
    }
    else
    {
        cout << "Stack cheia!" << endl;
    }
}

ItemType pop(Stack &p)
{
    if (!isEmpty(p))
    {
        ItemType tmp = p.data[p.length - 1];
        p.length--;
        return tmp;
    }
    else
    {
        cout << "Stack vazia!" << endl;
        return false;
    }
}

void describe(Stack p)
{
    if (isEmpty(p))
    {
        cout << "Stack: vazia" << endl;
        return;
    }

    cout << "Stack: ";
```

```
    for (int i = 0; i < p.length; i++)
    {
        cout << p.data[i] << " ";
    }
    cout << endl;
}

/* MEMORY SECTION */
float Memory[MEM_SIZE];

/* Verificar se os caracteres são numeros e inclui outras validações */
bool isNumber(const string& str)
{

    /* Verifica se é operação de subtração e somente espaço */
    if (str.length() == 1 && str[0] == '-' || str.length() == 0)
        return false;

    /* Verificadores de duplicidade */
    bool haveDot = false;
    bool haveMinus = false;

    int count = 0;

    // for tipo Lua.
    for (char const &c : str) {

        // tem dois . ou - retorna false
        if (c == '.' && haveDot) return false; // tem dois .
        if (c == '-' && haveMinus) return false; // tem dois -
        if (c == '.') haveDot = true;
        if (c == '-') haveMinus = true;

        // nao é numero nem ponto
        if ( (c < 48 || c > 57) && (c != 46) && (c != 45)) return false;

        // ha um - fora da posicao 0
        if (c==45 && count > 0) return false;

        count++;
```

```
    }

    return true;
}

/* Verifica se é um comando válido para memoria */
bool isMemoryCommand(string s)
{

    if ( (s.length() >= 3) && (s.length() <= 4) && (s.at(0) != ' ') && (s.at(1)
== ' ') && (s.at(2) != ' '))
    {
        if(s.at(0) != 'r' && s.at(0) != 's')
            return false;
        return true;
    }

    return false;
}

/* Tenta executar o comando de memoria */
bool doMemoryCommand(string s, Stack &p)
{

    string n = s.substr(2, s.length()-1);

    int pos;

    if (isNumber(n))
    {
        pos = std::stoi(n);

        if (pos <= 0 || pos > MEM_SIZE )
        {
            cout << "\tPosicao  invalido, use [r|s] [1-"<<MEM_SIZE<<"]\n";
            return false;
        }

    }else{
```

```
        cout << "\tFormato invalido, use [r|s] [1-"<<MEM_SIZE<<"]\n";
        return false;
    }

    if (s.at(0) == 's')
    {
        if (isEmpty(p))
        {
            cout << "\tNao ha valores na calculadora." << endl;
            return false;
        }

        Memory[pos-1] = pop(p);
        cout << "\tValor '"<< Memory[pos-1] <<"' salvo na memoria " << pos <<
endl;
    }
    else
    {
        if (isFull(p))
        {
            cout << "\tNao ha mais espaco na calculadora." << endl;
            return false;
        }

        push(p, Memory[pos-1]);
        cout << "\tValor '"<< Memory[pos-1] <<"' recuperado da memoria " << pos
<< endl;
        Memory[pos-1] = 0;
    }

    return true;
}

/* NPR */
// my registers, to serve as cpu registers, and avoid DRY
float ax, bx;

/* Idermediario para a realização das aperações, coloca os valores nos
    registradores ax,bx caso tenha o suficiante para realizar a operação */
bool stackOperandLenghtLtTwo(Stack &p)
```

```
{
    if (p.length < 2)
    {
        cout << "\tNao ha operadores suficientes para realizar a operacao\n";
        return false;
    }

    ax = pop(p);
    bx = pop(p);

    return true;
}

/* Idermediario para a realização das aperações, coloca os valores nos
    registradores ax,bx caso tenha o suficiante para realizar a operação */
bool stackOperandLenghtLtOne(Stack &p)
{
    if (p.length < 1)
    {
        cout << "\tNao ha operadores suficientes para realizar a operacao\n";
        return false;
    }

    ax = pop(p);

    return true;
}

/* Retorna a igualdade de duas strings */
bool stringEquals(string a, string b)
{
    return a.compare(b) == 0;
}

/* Uma simples função de fatorial, sem muitas verificações
    retorna o valor inteiro do fatorial . Ex: 5.x => 120
*/
float fatorial(float n)
{
    float fac = 1;
```

```
    if (n < 0)
    {
        cout << "\t Nao ha fatorial de numero negativo." << endl;
        return n;
    }
    else {
        for(int i = n; i > 1; i--) {
            fac *= i;
        }
    }
    return fac;

}

/* Exibe o menu de instruções */
void instructions()
{
    cout << " ====== Intrucoes ====== " << endl;
    cout << "Use 'h' para exibir o menu de ajuda" << endl;
    cout << "Use 'p' para ver a pilha atual" << endl;
    cout << "Use '=' para ver o valor atual" << endl;
    cout << "Escreva 'del' para apagar o ultimo valor inserido" << endl;
    cout << "Escreva 'reset' para apagar todos os valores inseridos" << endl;
    cout << "Escreva 's 3' para salvar o valor atual na memoria 3" << endl;
    cout << "Escreva 'r 7' para recuperar o valor atual da memoria 7" << endl;
    cout << "Posicoes disponiveis de 1-"<< MEM_SIZE << " para salvar e recuperar
valores da memoria" << endl;
    cout << "Escreva 'mem' para ver todos os valores salvos na memoria." <<
endl;
    cout << "====== Operacoes basicas ====== " << endl;
    cout << "Operacoes basicas: + - * / " << endl;
    cout << "====== Constantes ====== " << endl;
    cout << " 'pi' => "<< PI << endl;
    cout << "====== Operacoes extras ====== " << endl;
    cout << "//  (divisao inteira)  abs (valor absoluto)  | ^ (pontencia)  | !
(fatorial) " << endl;
    cout << "log (log de a na base b) | raiz (raiz de a no grau b)" << endl;
    cout << "sen, cos, tan | todos em radianos" << endl;
    cout << "asen, acos, atan | todos em radianos" << endl;
```

```
    cout << "Escreva 'exit' para encerrar a aplicacao"  << endl;
}

/* Exibe o valor do topo da pilha */
void showStackTopAsResult(Stack p)
{
    cout << "\t= ";
    cout << (float) p.data[p.length-1];
    cout<< endl;
}

int main()
{
    Stack operands;
    string op = "";

    instructions();

    // set os valores da Memory para 0
    for (int i = 0; i < MEM_SIZE; i++) { Memory[i] = 0;}

    while (!stringEquals(op,"exit"))
    {
        cout << "> ";
        getline(cin, op);

        // verificar se é numero decimal e se a stack está cheia
        if (isNumber(op))
        {
            if (isFull(operands))
            {
                cout << "\tNao ha mais espaco na calculadora." << endl;
            }
            else
            {
                push(operands, std::stof(op)); // converte pra float e da push
            }
        }
        /* Verificação de constantes */
        else if (stringEquals(op,"pi"))
```

```
        {
            push(operands,PI);
        }
         /* Comandos */
        else if (stringEquals(op,"exit"))
        {
            cout << "Programa encerrado." << endl;
        }
        else if (stringEquals(op,"p"))
        {
            cout << "Atual ";
            describe(operands);
            cout << endl;
        }
        else if (stringEquals(op,"h"))
        {
            instructions();
        }
        else if (stringEquals(op,"="))
        {
            cout << "\tR: ";
            cout << (float) operands.data[operands.length-1];
            cout<< endl;
        }
        else if (stringEquals(op,"del"))
        {
            if (!isEmpty(operands))
                cout << "\tValor: " << pop(operands) << " apaguado." << endl;
            else
                cout << "\tNao ha valores na calculadora." << endl;
        }
        else if (stringEquals(op,"reset"))
        {
            while (!isEmpty(operands))
            {
                pop(operands);
            }
            cout << "\tMemoria da calculadora resetada." << endl;
        }
        else if (stringEquals(op,"mem"))
```

```
        {    cout << "\t Memory: ";
            for (int i = 0; i < MEM_SIZE; i++) {cout << Memory[i] << " ";}
            cout << endl;
        }
        else if (isMemoryCommand(op))
        {
            doMemoryCommand(op, operands);
        }

        /* operações binarias */
        else if(stringEquals(op,"+"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                push(operands, ax+bx);
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"*"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                push(operands, ax*bx);
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"^"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                push(operands, pow(bx,ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"/"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                /* Verifica div 0 */
                if (ax == 0.0)
```

```
            {
                /* Solta uma camapainha \a como som de erro */
                cout << "\t Nao e possivel dividir por zero! \a" << endl;
                // devolve os valores para stack
                push(operands,bx);
                push(operands,ax);
            }else
            {
                push(operands, bx/ax);
            }
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"//"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            /* Verifica div 0 */
            if (ax == 0.0)
            {
                /* Solta uma camapainha \a como som de erro */
                cout << "\t Nao e possivel dividir por zero! \a" << endl;
                // devolve os valores para stack
                push(operands,bx);
                push(operands,ax);
            }else
            {
                push(operands, (int) (bx/ax));
            }
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"%"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            /* Verifica div 0 */
            if (ax == 0.0)
            {
                /* Solta uma camapainha \a como som de erro */
```

```
                cout << "\t Nao e possivel dividir por zero! \a" << endl;
                // devolve os valores para stack
                push(operands,bx);
                push(operands,ax);
            }else
            {
                // pois é nao aceita bx % ax
                push(operands,  fmod(bx,ax));
            }
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"log"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            if (ax == 0.0)
            {
                /* Solta uma camapainha \a como som de erro */
                cout << "\t Nao e possivel operar com zero! \a" << endl;
                // devolve os valores para stack
                push(operands,bx);
                push(operands,ax);
            }
            else
            {
                push(operands,  log(bx)/log(ax));
                showStackTopAsResult(operands);
            }
        }
    }
    else if (stringEquals(op,"-"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            push(operands, bx-ax);
            showStackTopAsResult(operands);
        }
    }
```

```
        /* operações unarias */
        else if (stringEquals(op,"!"))
        {
            if (stackOperandLenghtLtOne(operands))
            {


                push(operands, fatorial(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"sen"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, sin(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"cos"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, cos(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"tan"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, tan(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"asen"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, asin(ax));
```

```
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"acos"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, acos(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"atan"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, atan(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"abs"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, abs(ax));
                showStackTopAsResult(operands);
            }
        }
        else
        {
            cout << "Operacao invalida..." << endl;
        }
    }

    return 0;
}
```

2.1.4 – Implementação da Calculadora Polonesa Reversa com lista encadeada

```
#include <iostream>
#include <cmath>
using namespace std;

/*  Autor: Kevin Souza
    Calculadora Polonesa Inversa v4.0

    Uma calculadora feita para ser utilizada de verdade, por isso
    ela vem com vários recursos extras. Além de realizar operações matemáticas
    básicas ela possui uma pequena memória para armazanamento de valores.
    Implementação com lista encadeada.

*/
#define PI 3.14159265

/* Tamanho máximo das memórias de numeros */
#define MEM_SIZE 10

/* Atalho de tipagem genérica */
typedef float ItemType;

#include <iostream>
using namespace std;

struct Node{
    ItemType v;
    Node* next = NULL;
};

struct Stack{
    Node* node = NULL;
};

bool isEmpty(Stack *p)
{
    return (p->node == NULL);
}
```

```
bool isFull(Stack *p)
{
    try
    {
        Node *newnode = new Node;
        delete newnode;
        return false;
    }
    catch(std::bad_alloc e)
    {
        return true;
    }

}

void push(Stack* &p, ItemType n)
{
    if(!isFull(p))
    {
        Node *newnode = new Node;

        newnode->v = n;
        newnode->next = p->node;

        p->node = newnode;
    }
    else
    {
        cout << "Stack cheia! " << endl;
    }


}

ItemType pop(Stack* &p)
{
    if(!isEmpty(p))
    {
```

```
        Node *tmpnode  = p->node;
        ItemType tmp = tmpnode->v;
        p->node = p->node->next;
        delete tmpnode;

        return tmp;
    }
    else
    {
        cout << "Stack vazia!" << endl;
        ItemType n;
        return n;
    }

}

/* Retorna a quantidade de itens na stack */
int stackLenght(Stack* p)
{
    int tam = 0;

    for(Node *aux = p->node; aux != NULL; aux = aux->next)
        tam++;

    return tam;
}

/**
 * Função que exibe na tela uma representação da pilha.
 */
void describe(Stack *p)
{
    Node *aux = p->node;

    if (aux == NULL)
    {
        cout << "Stack Vazia" << endl;
        delete aux;
    }else
    {
```

```
        cout << "Stack: " ;
        for(Node *aux2 = p->node; aux2 != NULL; aux2 = aux2->next)
        {
            cout << aux2->v << " ";
        }
        cout << endl;
    }
}


/* MEMORY SECTION */
float Memory[MEM_SIZE];

/* Verificar se os caracteres são numeros e inclui outras validações */
bool isNumber(const string& str)
{

    /* Verifica se é operação de subtração e somente espaço */
    if (str.length() == 1 && str[0] == '-' || str.length() == 0)
        return false;

    /* Verificadores de duplicidade */
    bool haveDot = false;
    bool haveMinus = false;

    int count = 0;

    // for tipo Lua.
    for (char const &c : str) {

        // tem dois . ou - retorna false
        if (c == '.' && haveDot) return false; // tem dois .
        if (c == '-' && haveMinus) return false; // tem dois -
        if (c == '.') haveDot = true;
        if (c == '-') haveMinus = true;

        // nao é numero nem ponto
        if ( (c < 48 || c > 57) && (c != 46) && (c != 45)) return false;

        // ha um - fora da posicao 0
```

```
        if (c==45 && count > 0) return false;

        count++;
    }

    return true;
}

/* Verifica se é um comando válido para memoria */
bool isMemoryCommand(string s)
{

    if ( (s.length() >= 3) && (s.length() <= 4) && (s.at(0) != ' ') && (s.at(1)
== ' ') && (s.at(2) != ' '))
    {
        if(s.at(0) != 'r' && s.at(0) != 's')
            return false;
        return true;
    }

    return false;
}

/* Tenta executar o comando de memoria */
bool doMemoryCommand(string s, Stack* p)
{

    string n = s.substr(2, s.length()-1);

    int pos;

    if (isNumber(n))
    {
        pos = std::stoi(n);

        if (pos <= 0 || pos > MEM_SIZE )
        {
            cout << "\tPosicao  invalido, use [r|s] [1-"<<MEM_SIZE<<"]\n";
            return false;
        }
```

```
    }else{

        cout << "\tFormato invalido, use [r|s] [1-"<<MEM_SIZE<<"]\n";
        return false;
    }

    if (s.at(0) == 's')
    {
        if (isEmpty(p))
        {
            cout << "\tNao ha valores na calculadora." << endl;
            return false;
        }

        Memory[pos-1] = pop(p);
        cout << "\tValor '"<< Memory[pos-1] <<"' salvo na memoria " << pos <<
endl;
    }
    else
    {
        if (isFull(p))
        {
            cout << "\tNao ha mais espaco na calculadora." << endl;
            return false;
        }

        push(p, Memory[pos-1]);
        cout << "\tValor '"<< Memory[pos-1] <<"' recuperado da memoria " << pos
<< endl;
        Memory[pos-1] = 0;
    }

    return true;
}

/* NPR */
// my registers, to serve as cpu registers, and avoid DRY
float ax, bx;
```

```
/* Idermediario para a realização das aperações, coloca os valores nos
    registradores ax,bx caso tenha o suficiante para realizar a operação */
bool stackOperandLenghtLtTwo(Stack* p)
{
    if (stackLenght(p) < 2)
    {
        cout << "\tNao ha operadores suficientes para realizar a operacao\n";
        return false;
    }

    ax = pop(p);
    bx = pop(p);

    return true;
}

/* Idermediario para a realização das aperações, coloca os valores nos
    registradores ax,bx caso tenha o suficiante para realizar a operação */
bool stackOperandLenghtLtOne(Stack *p)
{
    if (stackLenght(p) < 1)
    {
        cout << "\tNao ha operadores suficientes para realizar a operacao\n";
        return false;
    }

    ax = pop(p);

    return true;
}

/* Retorna a igualdade de duas strings */
bool stringEquals(string a, string b)
{
    return a.compare(b) == 0;
}

/* Uma simples função de fatorial, sem muitas verificações
    retorna o valor inteiro do fatorial . Ex: 5.x => 120
*/
```

```
float fatorial(float n)
{
    float fac = 1;

    if (n < 0)
    {
        cout << "\t Nao ha fatorial de numero negativo." << endl;
        return n;
    }
    else {
        for(int i = n; i > 1; i--) {
            fac *= i;
        }
    }
    return fac;

}

/* Exibe o menu de instruções */
void instructions()
{
    cout << " ====== Intrucoes ====== " << endl;
    cout << "Use 'h' para exibir o menu de ajuda" << endl;
    cout << "Use 'p' para ver a pilha atual" << endl;
    cout << "Use '=' para ver o valor atual" << endl;
    cout << "Escreva 'del' para apagar o ultimo valor inserido" << endl;
    cout << "Escreva 'reset' para apagar todos os valores inseridos" << endl;
    cout << "Escreva 's 3' para salvar o valor atual na memoria 3" << endl;
    cout << "Escreva 'r 7' para recuperar o valor atual da memoria 7" << endl;
    cout << "Posicoes disponiveis de 1-"<< MEM_SIZE << " para salvar e recuperar
valores da memoria" << endl;
    cout << "Escreva 'mem' para ver todos os valores salvos na memoria." <<
endl;
    cout << "====== Operacoes basicas ====== " << endl;
    cout << "Operacoes basicas: + - * / " << endl;
    cout << "====== Constantes ====== " << endl;
    cout << " 'pi' => "<< PI << endl;
    cout << "====== Operacoes extras ====== " << endl;
    cout << "//  (divisao inteira)  abs (valor absoluto)  | ^ (pontencia)  | !
(fatorial) " << endl;
```

```
    cout << "log (log de a na base b) | raiz (raiz de a no grau b)" << endl;
    cout << "sen, cos, tan | todos em radianos" << endl;
    cout << "asen, acos, atan | todos em radianos" << endl;
    cout << "Escreva 'exit' para encerrar a aplicacao"  << endl;
}

/* Exibe o valor do topo da pilha */
void showStackTopAsResult(Stack* p)
{
    cout << "\t= ";
    cout << (float) p->node->v;
    cout<< endl;
}

int main()
{
   //criação da pilha com o nome operands
    Stack *operands = new Stack;

    string op = "";

    instructions();

    // set os valores da Memory para 0
    for (int i = 0; i < MEM_SIZE; i++) { Memory[i] = 0;}

    while (!stringEquals(op,"exit"))
    {
        cout << "> ";
        getline(cin, op);

        // verificar se é numero decimal e se a stack está cheia
        if (isNumber(op))
        {
            if (isFull(operands))
            {
                cout << "\tNao ha mais espaco na calculadora." << endl;
            }
            else
            {
```

```
            push(operands, std::stof(op)); // converte pra float e da push
        }
    }
    /* Verificação de constantes */
    else if (stringEquals(op,"pi"))
    {
        push(operands,PI);
    }
     /* Comandos */
    else if (stringEquals(op,"exit"))
    {
        cout << "Programa encerrado." << endl;
    }
    else if (stringEquals(op,"p"))
    {
        cout << "Atual ";
        describe(operands);
        cout << endl;
    }
    else if (stringEquals(op,"h"))
    {
        instructions();
    }
    else if (stringEquals(op,"="))
    {
        cout << "\tR: ";
        cout << (float) operands->node->v;
        cout<< endl;
    }
    else if (stringEquals(op,"del"))
    {
        if (!isEmpty(operands))
            cout << "\tValor: " << pop(operands) << " apaguado." << endl;
        else
            cout << "\tNao ha valores na calculadora." << endl;
    }
    else if (stringEquals(op,"reset"))
    {
        while (!isEmpty(operands))
        {
```

```
            pop(operands);
        }
        cout << "\tMemoria da calculadora resetada." << endl;
    }
    else if (stringEquals(op,"mem"))
    {   cout << "\t Memory: ";
        for (int i = 0; i < MEM_SIZE; i++) {cout << Memory[i] << " ";}
        cout << endl;
    }
    else if (isMemoryCommand(op))
    {
        doMemoryCommand(op, operands);
    }

    /* operações binarias */
    else if(stringEquals(op,"+"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            push(operands, ax+bx);
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"*"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            push(operands, ax*bx);
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"^"))
    {
        if (stackOperandLenghtLtTwo(operands))
        {
            push(operands, pow(bx,ax));
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"/"))
```

```
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                /* Verifica div 0 */
                if (ax == 0.0)
                {
                    /* Solta uma camapainha \a como som de erro */
                    cout << "\t Nao e possivel dividir por zero! \a" << endl;
                    // devolve os valores para stack
                    push(operands,bx);
                    push(operands,ax);
                }else
                {
                    push(operands, bx/ax);
                }
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"//"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                /* Verifica div 0 */
                if (ax == 0.0)
                {
                    /* Solta uma camapainha \a como som de erro */
                    cout << "\t Nao e possivel dividir por zero! \a" << endl;
                    // devolve os valores para stack
                    push(operands,bx);
                    push(operands,ax);
                }else
                {
                    push(operands, (int) (bx/ax));
                }
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"%"))
        {
            if (stackOperandLenghtLtTwo(operands))
```

```
            {
                /* Verifica div 0 */
                if (ax == 0.0)
                {
                    /* Solta uma camapainha \a como som de erro */
                    cout << "\t Nao e possivel dividir por zero! \a" << endl;
                    // devolve os valores para stack
                    push(operands,bx);
                    push(operands,ax);
                }else
                {
                    // pois é nao aceita bx % ax
                    push(operands,  fmod(bx,ax));
                }
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"log"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
                if (ax == 0.0)
                {
                    /* Solta uma camapainha \a como som de erro */
                    cout << "\t Nao e possivel operar com zero! \a" << endl;
                    // devolve os valores para stack
                    push(operands,bx);
                    push(operands,ax);
                }
                else
                {
                    push(operands,  log(bx)/log(ax));
                    showStackTopAsResult(operands);
                }
            }
        }
        else if (stringEquals(op,"-"))
        {
            if (stackOperandLenghtLtTwo(operands))
            {
```

```
            push(operands, bx-ax);
            showStackTopAsResult(operands);
        }
    }

    /* operações unarias */
    else if (stringEquals(op,"!"))
    {
        if (stackOperandLenghtLtOne(operands))
        {


            push(operands, fatorial(ax));
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"sen"))
    {
        if (stackOperandLenghtLtOne(operands))
        {
            push(operands, sin(ax));
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"cos"))
    {
        if (stackOperandLenghtLtOne(operands))
        {
            push(operands, cos(ax));
            showStackTopAsResult(operands);
        }
    }
    else if (stringEquals(op,"tan"))
    {
        if (stackOperandLenghtLtOne(operands))
        {
            push(operands, tan(ax));
            showStackTopAsResult(operands);
        }
    }
```

```
        else if (stringEquals(op,"asen"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, asin(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"acos"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, acos(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"atan"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, atan(ax));
                showStackTopAsResult(operands);
            }
        }
        else if (stringEquals(op,"abs"))
        {
            if (stackOperandLenghtLtOne(operands))
            {
                push(operands, abs(ax));
                showStackTopAsResult(operands);
            }
        }
        else
        {
            cout << "Operacao invalida..." << endl;
        }
    }
    return 0;
}
```