Logotipo

Descrição gerada automaticamente

UNIVERSIDADE FEDERAL DO AMAZONAS

FACULDADE DE TECNOLOGIA

ENGENHARIA DA COMPUTAÇÃO

LABORATÓRIO 1 - PREPARAÇÃO DO AMBIENTE

Daniel Santiago da Silva -21950879

MANAUS-AM

2022

Daniel Santiago da Silva - 21950879

LAB 01: SIMULAÇÃO DE SISTEMAS E REALIMENTAÇÃO

Primeiro Relatório da disciplina de Programação de Sistemas de Tempo Real referente à preparação do ambiente de trabalho que será avaliado pelo professor André Cavalcante junto ao Curso de Engenharia da Computação.

Professor: André Cavalcante

MANAUS-AM

2022

**SUMÁRIO**

1. **OBJETIVOS ...........................................................................................2**
2. **INTRODUÇÃO TEÓRICA ......................................................................3**
3. **ESTRUTURA E ORGANIZAÇÃO DOS DIRETÓRIOS .........................4**
   1. ARQUIVOS FONTES E ESTRUTURA DE DIRETÓRIOS **.................4**
   2. MAKEFILE **..........................................................................................5**
4. **ADTS .......................................................................................................6**
   1. DYNAMIC STRING**..............................................................................6**
   2. INTEGRAL**.........................................................................................13**
   3. MATRIX**.............................................................................................16**
5. **CONCLUSÃO ......................................................................................30**
6. **REFERÊNCIAS ...................................................................................31**
7. **OBJETIVOS**

Os objetivos desse primeiro laboratório de Programação de Sistemas de Tempo Real envolvem o desenvolvimento de ADTs de suporte além da configuração de um ambiente de trabalho utilizando a linguagem C padrão 17.

As ADTs criadas neste trabalho são a de operações matriciais em ponto flutuante, a de integração de funções e a de criação de strings dinâmicas. Essas bibliotecas terão fundamental importância nos trabalhos que serão feitos ao decorrer do curso.

1. **INTRODUÇÃO TEÓRICA - ADT**

As linguagens de programação têm suas ferramentas e métodos que nos permitem criar tipos de dados além dos primitivos contidos na própria linguagem. Esses tipos são chamados de ADTs (Abstract Data Types) e estão em constante uso na implementação de diversas soluções no mundo. Além de especificar as informações contidas nesses dados, as ADTs envolvem as operações, também conhecidas como métodos, que as manipulam.

Esses métodos são definidos em uma interface conhecida como API (Applications Programming Interface), contudo sua implementação ocorre em outra parte do código. Essa implementação deve levar em conta os objetivos iniciais para a criação da ADT, focando na característica de desejo: desempenho, flexibilidade ou qualquer outro parâmetro de interesse.

Os usuários do dos métodos da ADT são chamados de clientes, contudo seu acesso é limitado apenas a API da abstração, ou seja, a interface tem o papel de dar ao usuário o acesso aos métodos da ADT.

1. **ESTRUTURA E ORGANIZAÇÃO DOS DIRETÓRIOS**
   1. ARQUIVOS FONTESE ESTRUTURA DE DIRETÓRIOS

O trabalho segue a seguinte organização dos diretórios, na pasta raiz há 3 diretórios: as pastas *src* (arquivos fonte), *inc* (interfaces/*includes*) e *obj* (objetos). Além delas, o diretório raiz contém o arquivo Makefile e o executável do programa com o mesmo nome do diretório raiz, nesse caso “PTR-Entrega”.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Figura 1: Diretórios do trabalho

Entre os arquivos fonte da pasta *src* temos as implementações das ADTs (DString, Matrix e Integral) com o código de todas as suas operações no formato “*.c*”. O arquivo que contém a função main com os testes das ADTs também se encontra nesse diretório.

Na pasta *inc* se observa os arquivos “*.h*” do projeto, ou seja, as APIs ou interfaces. Neles há apenas as definições das operações implementadas nos arquivos fontes e estão separados em um arquivo para cada ADT (dstring.h, integrate.h e matrix.h).

O diretório *obj* armazena os objetos gerados nas compilações, da mesma forma que acontece com a interface é há um arquivo “*.o*” para cada ADT (dstring.o, integrate.o e matrix.o). Os arquivos objetos são as saídas reais geradas na fase de compilação, comumente escrita em formato binário ou linguagem de máquina, nesse caso eles são gerados por conta da compilação no *gcc* utilizando o parâmetro *-c* que significa “*just compile*” (apenas compile).

Por fim, no diretório raiz se encontra o arquivo Makefile responsável por simplificar o processo de compilar e gerar o arquivo executável que por sua vez também está nesse diretório. Dessa forma para se executar os testes das operações implementadas na pasta src e definidas na pasta inc basta executar o comando make no diretório raiz e em seguida abrir o arquivo executável gerado ./PTR-Entrega.

As ferramentas utilizadas no desenvolvimento do laboratório foram o editor de texto *VSCode*, a compilação em C foi feita em máquina virtual (WSL) contendo o Ubuntu 20.04 e *gcc* instalado pelo *build-essentials*.

* 1. MAKEFILE

Texto

Descrição gerada automaticamenteComo descrito anteriormente, o arquivo Makefile é muito utilizado para se automatizar o processo de compilação dos arquivos fontes e seus *includes*. Neles são escritos todos os parâmetros e argumentos desejados na hora da compilação além de se informar os diretórios dos códigos do programa. O Makefile utilizado foi o mesmo disponibilizado nos exemplos de ADTs desenvolvido pelo professor André Cavalcante retirando apenas as variáveis relacionadas a outras bibliotecas de um diretório *lib*. Note que executável de teste possui o mesmo nome do diretório raiz do programa.

Figura 2: Comandos gerado pelo Makefile.

1. **ADTS**
   1. DYNAMIC STRING

Esse tipo abstrato de dado tem como objetivo gerar *strings* com alocação dinâmica e é definida por um ponteiro de char que aponta para a *string* e um número inteiro que indica quantos caracteres estão alocados no *buffer*. A *struct* que define a ADT bem como as definições das operações estão no arquivo *dstring.h.*

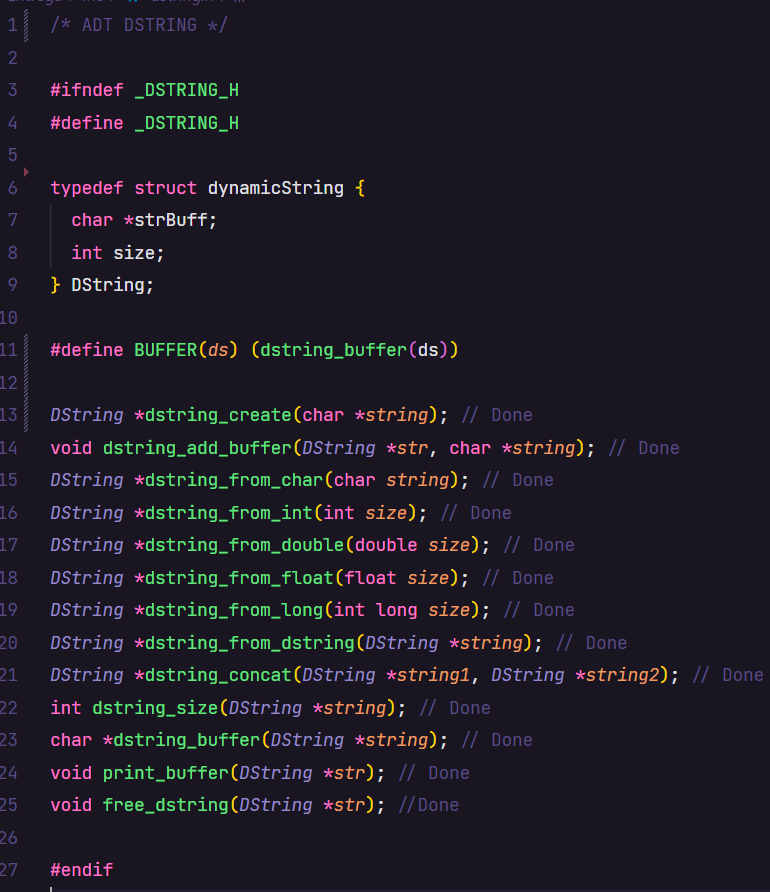


Figura 3: API *dstring.h*

As funções implementadas nessa ADT são as seguintes:

* ***dstring\_create:***O primeiro método desenvolvido foi a criação de uma string dinâmica a partir de um ponteiro do tipo *char*. A função recebe como parâmetro um ponteiro do tipo *char* e o percorre para definir a quantidade de caracteres contidos nele. Em seguida cria uma instância *DString* e atualiza o seu *buffer* com o ponteiro passado e o seu tamanho com o descoberto anteriormente. Seu retorno é o ponteiro para a instância criada do tipo *DString*.

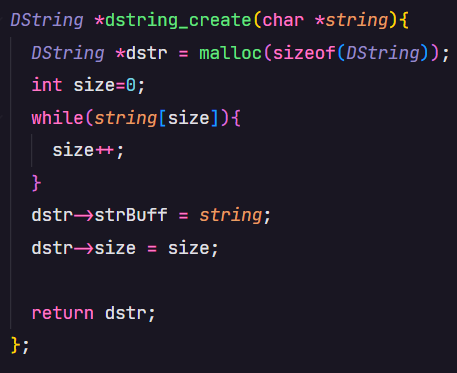


Figura 4: Implementação do *dstring\_create*

* ***dstring\_add\_buffer:***Esse método tem como função adicionar um *buffer* a uma *DString* com memória já alocada. Dessa forma ele recebe como parâmetro um ponteiro do tipo *DString* e um ponteiro do tipo *char*, em seguida copia os valores da *string* para o *buffer* da *DString* até que ocupe todo o espaço de memória alocado previamente ou até que os caracteres da *string* de entrada acabem. Como retorno temos o ponteiro para a *DString* contendo o novo *buffer.*

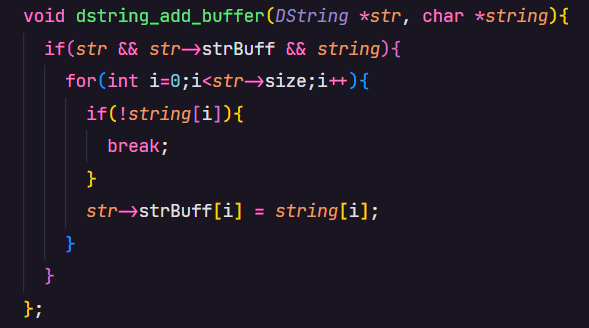
******

Figura 5: Implementação do *dstring\_add\_buffer*

* ***dstring\_size:*** Função que recebe o ponteiro de uma *DString* e retorna o valor salvo do seu tamanho alocado.

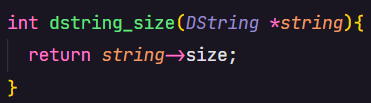
******

Figura 6: Implementação do *dstring\_size*

* ***dstring\_from\_int:*** Função que recebe um inteiro como parâmetro de entrada e cria uma instância do tipo *DString* com a memória alocada no *buffer* para a quantidade de caracteres informada na entrada. Retorna o ponteiro da *DString* criada.

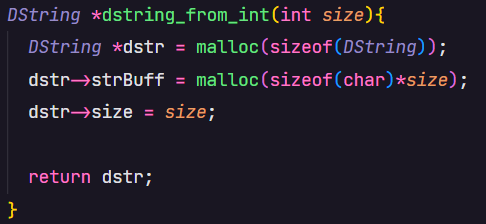
******

Figura 7: Implementação do *dstring\_from\_int*

* ***dstring\_from\_char:*** Função que recebe um *char* como parâmetro de entrada e cria uma instância do tipo *DString* com a memória alocada no *buffer* para esse *char* além de o copiar seu valor nele. Retorna o ponteiro da *DString* criada.

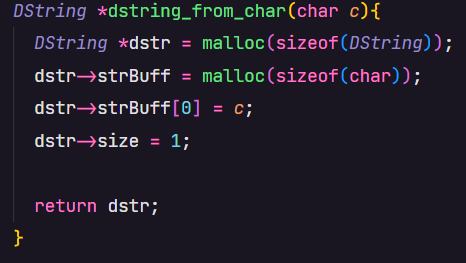
******

Figura 8: Implementação do *dstring\_from\_char*

* ***dstring\_from\_double:*** Função que recebe um ponto flutuante do tipo *double* como parâmetro de entrada e cria uma instância do tipo *DString* com a memória alocada no *buffer* para a quantidade de caracteres informada na entrada. O valor recebido é convertido em inteiro antes que a alocação de memória ocorra. Retorna o ponteiro da *DString* criada.

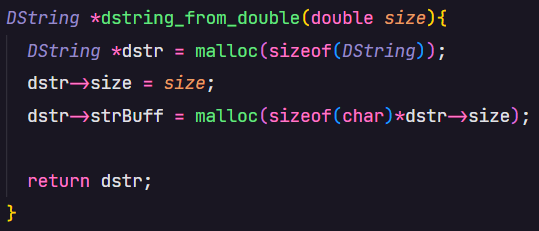
******

Figura 9: Implementação do *dstring\_from\_double*

* ***dstring\_from\_float:*** Função que recebe um ponto flutuante do tipo *float* como parâmetro de entrada e cria uma instância do tipo *DString* com a memória alocada no *buffer* para a quantidade de caracteres informada na entrada. O valor recebido é convertido em inteiro antes que a alocação de memória ocorra. Retorna o ponteiro da *DString* criada.

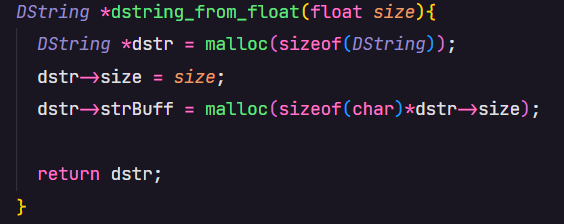
******

Figura 10: Implementação do *dstring\_from\_float*

* ***dstring\_from\_long:*** Função que recebe um inteiro do tipo *long* como parâmetro de entrada e cria uma instância do tipo *DString* com a memória alocada no *buffer* para a quantidade de caracteres informada na entrada. Retorna o ponteiro da *DString* criada.

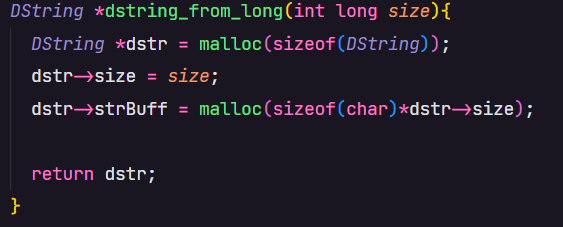
******

Figura 11: Implementação do *dstring\_from\_long*

* ***dstring\_from\_dstring:*** Esse método recebe uma *DString* como entrada e cria uma segunda *DString* copiando o tamanho alocado e o *buffer*. Note que esse método copia os valores da *string* dinâmica de entrada. Seu retorno é o ponteiro da *DString* gerada.

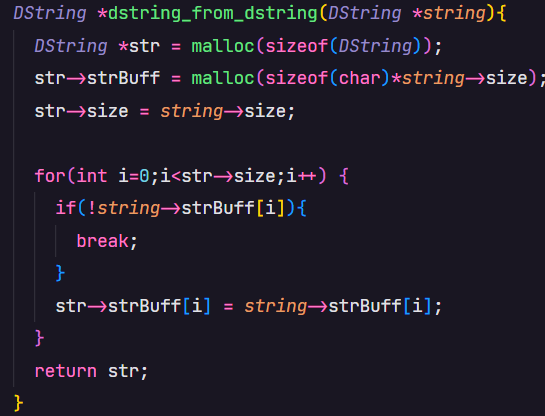
******

Figura 12: Implementação do *dstring\_from\_dstring*

* ***dstring\_concat:*** Esse operador tem como objetivo concatenar duas *DString*. Tem como parâmetros de entrada dois ponteiros do tipo *DString* e gera uma instância do mesmo tipo definindo como tamanho de caracteres de alocação a soma dos tamanhos contidos nas entradas. Para definir os valores do *buffer* da nova *DString* é realizada uma cópia dos caracteres contidos no *buffer* da primeira entrada e logo em seguida a cópia dos caracteres contidos no *buffer* da segunda entrada. O retorno da função é o ponteiro da *DString* gerado para a concatenação.

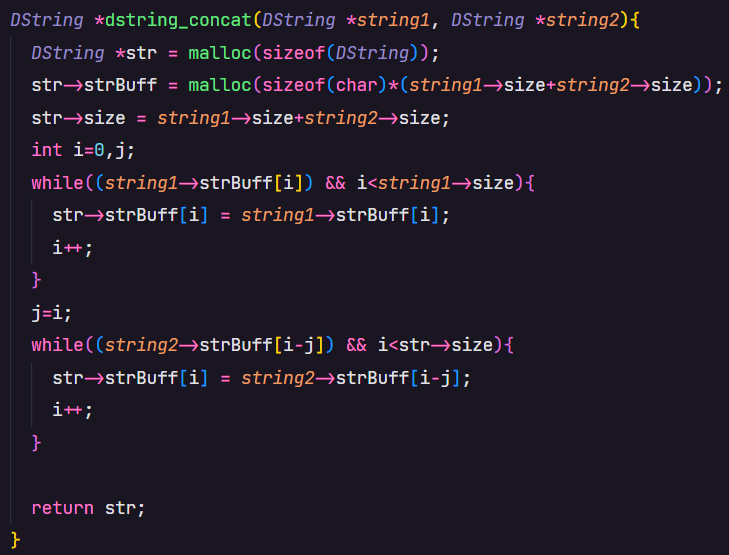
****

Figura 13: Implementação do *dstring\_concat*

* ***print\_buffer:*** Funçãoquerecebe um ponteiro do tipo *DString* e percorre todo o seu *buffer* imprimindo cada valor até que os caracteres terminem ou chegue no tamanho máximo alocado. Essa função não tem retorno.

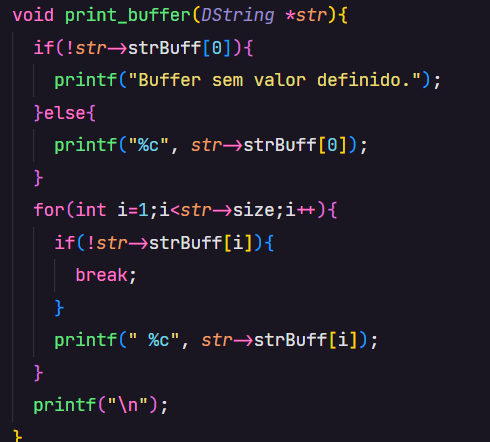
******

Figura 14: Implementação do *print\_buffer*

* ***dstring\_buffer:*** Operador que tem como função retornar o ponteiro *char* do *buffer* de uma instância do tipo *DString.* A entrada é o ponteiro dessa instância e o retorno é o ponteiro do *buffer.*

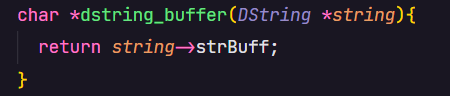
******

Figura 15: Implementação do *dstring\_buffer*

* ***free\_dstring:*** Função que limpa a memória do buffer de uma *DString* e zera o atributo de tamanho alocado. Seu parâmetro é o ponteiro para a *DString* a ser liberada. Não há retorno para esse método.

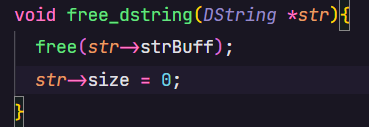
******

Figura 16: Implementação do *dstring\_free*

* ***BUFFER:*** Foi definido na API a função BUFFER que tem como parâmetro uma *DString* e executa a função *dstring\_buffer* descrita anteriormente.

******

Figura 17: Implementação do *BUFFER*

* 1. INTEGRAL

Essa ADT tem como objetivo de calcular a integral de funções em ponto flutuante. Sua definição é feita a partir de dois números em ponto flutuante que indicam o limite superior e inferior da integral além de um ponteiro para uma função que recebe *double* e retorna *double*. A interface presente no arquivo “*integrate.h”* é apresentada a seguir.

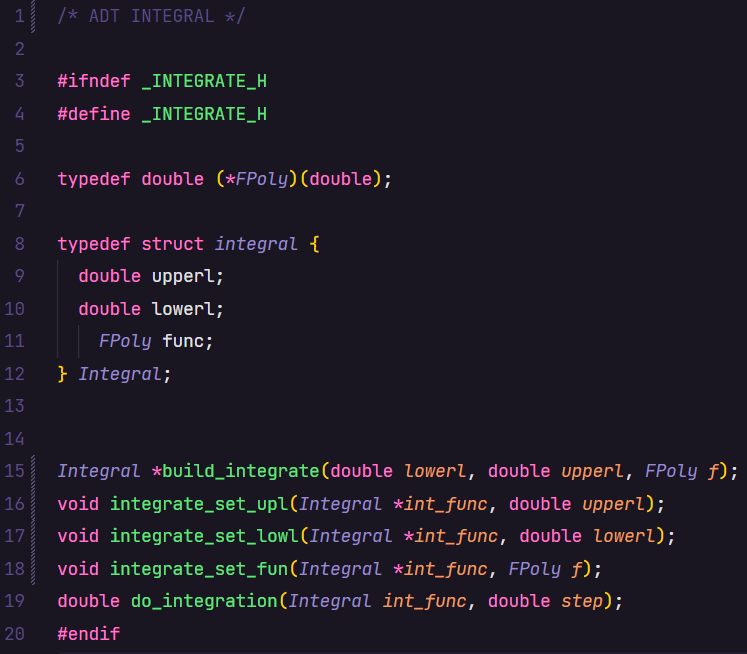


Figura 18: API *integrate.h*

As funções implementadas nessa ADT são as seguintes:

* ***build\_integrate:*** Essa função cria uma instância do tipo *Integral* a partir de dois parâmetros do tipo *double* que se referem aos limites superiores e inferiores da integral, além de um ponteiro da função a ser integrada. O retorno é o ponteiro para a instância *Integral* criada.

***Texto

Descrição gerada automaticamente***

Figura 19: Implementação do *build\_integrate*

* ***integrate\_set\_upl:*** Método que recebe um ponteiro do tipo *Integral* e um número do tipo *double* e o define como limite superior da integral.

***Texto

Descrição gerada automaticamente***

Figura 20: Implementação do *integrate\_set\_upl*

* ***integrate\_set\_lowl:*** Método que recebe um ponteiro do tipo *Integral* e um número do tipo *double* e o define como limite inferior da integral.

***Texto

Descrição gerada automaticamente***

Figura 21: Implementação do *integrate\_set\_lowl*

* ***integrate\_set\_fun:*** Método que recebe um ponteiro do tipo *Integral* e um ponteiro de uma função e o define como a função a ser integrada.

***Texto

Descrição gerada automaticamente***

Figura 22: Implementação do *integrate\_set\_fun*

* ***do\_integration:*** A operação que recebe um número do tipo *double* que indica a precisão a ser usada nos cálculos e uma instância do tipo Integral e executa a integração na função descrita nessa instância nos limites também contidos ali. O método utilizado para a implementação é o do trapézio composto descrito na fórmula a seguir:

***Tabela

Descrição gerada automaticamente***Eq. 1

Ao se executar esse método, é verificado se a instância do tipo *Integral* tem o ponteiro de uma função associado, se sim ele a executa no seu limite inferior e soma com o dobro das respostas da execução dessa função aumentando o seu parâmetro de execução de acordo com o passo descrito na entrada do *do\_integration*. Quanto menor for esse passo, maior será a precisão da resposta, contudo o custo computacional também aumenta já que o número de iterações cresce. Ao final o resultado é somado com a aplicação da função no limite superior e em seguida multiplicado pela metade do passo. O retorno é um número do tipo *double* com a aproximação do resultado da integral.

Tela preta com letras brancas

Descrição gerada automaticamente

Figura 23: Implementação *do\_integration*

* 1. MATRIZ

A ADT Matrix desenvolvida tem como objetivo realizar operações em matrizes com valores em ponto flutuante (*double*). Ela é definida pela estrutura que contém dois números inteiros indicando o número de linhas e colunas além de um ponteiro do tipo *double* que aponta para os valores da matriz.

Como base para esse desenvolvimento foi utilizada a ADT Matrix fornecida pelo professor André Cavalcante, contendo algumas operações básicas bem como a definição do tipo abstrato de dado. É importante salientar que algumas das funções fornecidas previamente apresentavam alguns erros como a função *matrix\_copy*, *matrix\_apply* e *matrix\_free* que foram corrigidos ao decorrer do trabalho.

Texto, chat ou mensagem de texto

Descrição gerada automaticamente

Figura 24: API com a definição e métodos disponibilizados previamente pelo professor André Cavalcante

Texto

Descrição gerada automaticamente

Figura 25: Métodos desenvolvidos no trabalho

Todas as funções disponibilizadas pela ADT no arquivo fonte serão explicadas a seguir:

* ***matrix\_value\_init:*** Método auxiliar que recebe um determinado número de linhas e colunas além de um valor específico e cria uma matriz com essas dimensões e a preenche com o valor especificado. Retorna a matriz criada.

***Tela preta com letras brancas

Descrição gerada automaticamente***

Figura 26: Implementação do *matrix\_value\_init*

* ***matrix\_zeros:*** Função que utiliza o método auxiliar anterior para gerar uma matriz preenchida pelo valor zero com as dimensões especificadas na entrada. O seu retorno é a matriz gerada.

***Texto

Descrição gerada automaticamente***

Figura 27: Implementação do *matrix\_zeros*

* ***matrix\_ones:*** Função que utiliza o método auxiliar *matrix\_value\_init* para gerar uma matriz preenchida pelo valor zero com as dimensões especificadas na entrada. O seu retorno é a matriz gerada.

***Texto

Descrição gerada automaticamente***

Figura 28: Implementação do *matrix\_ones*

* ***matrix\_identity:*** Esse método recebe dois inteiros que indicam o número de linhas e colunas, em seguida instancia uma matriz do tipo *Matrix*, define as dimensões dela a partir dos inteiros de entrada e preenche os valores com 0 quando a posição da linha for diferente da coluna e com 1 caso sejam iguais. O retorno é a matriz criada.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 29: Implementação do *matrix\_identity*

* ***matrix\_copy:*** Método que recebe uma matriz e a partir dela gera outra matriz com as mesmas dimensões, aloca memória para os valores de acordo com essa dimensão definida e percorre cada valor da matriz de entrada o copiando para a matriz gerada na mesma posição. Ao final a matriz gerada será uma cópia da que foi passada como parâmetro de entrada. O retorno é a matriz gerada, caso a entrada não possua valores o retorno será uma matriz nula.

***Texto

Descrição gerada automaticamente***

Figura 30: Implementação do *matrix\_copy*

* ***matrix\_apply:*** Este método recebe um ponteiro para uma função e uma matriz como entradas, em seguida gera uma matriz com as mesmas dimensões da matriz de entrada e define seus valores como a aplicação da função de entrada no valor da matriz de entrada em determinada posição. Caso a matriz de entrada não possua valores o retorno será uma matriz nula, caso tenha então o retorno será a matriz gerada.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 31: Implementação do *matrix\_apply*

* ***matrix\_free:*** Essa função recebe o ponteiro de uma matrix e libera a memória dos seus valores e em seguida fixa a sua dimensão em 0 e os valores em NULL.

***Texto

Descrição gerada automaticamente***

Figura 32: Implementação do *matrix\_free*

* ***VALUES:*** No arquivo fonte dessa ADT é definido o método *VALUES* que recebe uma matriz e uma posição de linha e coluna e retorna o valor da matriz contido nessa posição.



Figura 33: Implementação do *VALUES*

* ***matrix\_value:*** Função que recebe uma matriz e uma posição em linha e coluna retornando o valor da matriz contido nessa posição. É utilizado o método *VALUES* definido anteriormente.

***Texto

Descrição gerada automaticamente***

Figura 34: Implementação do *matrix\_value*

* ***matrix\_getValue:*** Método que recebe uma matriz e uma posição de linha e coluna e retorna o valor da matriz contido nessa posição.

Texto

Descrição gerada automaticamente

Figura 35: Implementação do *matrix\_getValue*

* ***matrix\_nlins:*** Funçãoque recebe uma matriz e retorna o seu número de linhas.

***Texto

Descrição gerada automaticamente***

Figura 36: Implementação do *matrix\_nlins*

* ***matrix\_ncols:*** Funçãoque recebe uma matriz e retorna o seu número de colunas.

***Texto

Descrição gerada automaticamente***

Figura 37: Implementação do *matrix\_ncols*

* ***matrix\_create***: Esse método tem como objetivo criar uma matriz a partir das dimensões e valores informados pelo usuário. Assim, ao ser executado é solicitado que seja inserido no terminal as dimensões da matriz (número de linhas e colunas), se forem número inteiros positivos maiores que zero então uma matriz é gerada e tem uma quantidade de memória alocada definida pelas dimensões informadas. Em seguida é perguntado do usuário qual o valor que se deseja inserir em cada posição da matriz. Caso algum erro aconteça, a matriz nula é retornada. Se não ocorrer nenhum problema então a matriz gerada é retornada.

***Texto

Descrição gerada automaticamente***

Figura 38: Implementação do *matrix\_create*

* ***matrix\_scalar\_mul:*** Essa função recebe um número do tipo *double* e uma matriz e caso ela tenha valores uma nova matriz com as mesmas dimensões é criada, se não houver então é retornada uma matriz nula. Em seguida a matriz de entrada tem seus valores percorridos e multiplicados pelo número especificado na entrada e o resultado é salvo na nova matriz. Por fim é retornado a matriz gerada.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 39: Implementação do *matrix\_scalar\_mul*

* ***matrix\_scalar\_sum:*** Essa função recebe um número do tipo *double* e uma matriz e caso ela tenha valores uma nova matriz com as mesmas dimensões é criada, se não houver então é retornada uma matriz nula. Em seguida a matriz de entrada tem seus valores percorridos e somados com número especificado na entrada e o resultado é salvo na nova matriz. Por fim é retornado a matriz gerada.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 40: Implementação do *matrix\_scalar\_sum*

* ***matrix\_scalar\_dif:*** Essa função recebe um número do tipo *double* e uma matriz e caso ela tenha valores uma nova matriz com as mesmas dimensões é criada, se não houver então é retornada uma matriz nula. Em seguida a matriz de entrada tem seus valores percorridos e subtraídos pelo número especificado na entrada e o resultado é salvo na nova matriz. Por fim é retornado a matriz gerada.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 41: Implementação do *matrix\_scalar\_dif*

* ***matrix\_mul:*** Nesta função é passada duas matrizes como parâmetros (*Matrix* A e *Matrix* B). Em seguida é verificado se a quantidade de colunas da matriz A é igual a quantidade de linhas da matriz B e se ambas as matrizes estão vazias, se as dimensões forem diferentes ou uma das matrizes estiver vazia é retornado uma matriz nula. Se passarem pela verificação uma nova matriz é criada com o número de linhas da matriz A e o número de colunas da matriz B. Como passo seguinte, o valor da nova matriz em uma posição (x,y) é definido como o somatório do produto entre os valores da linha x da matriz A e da coluna y da matriz B. Repetindo esse processo para todas as posições da nova matriz a retornaremos como resultado da função.

***Texto

Descrição gerada automaticamente***

Figura 42: Implementação do *matrix\_mul*

* ***matrix\_sum:*** Essa operação recebe como entrada duas matrizes na qual é verificada se possuem as mesmas dimensões e se elas não estão vazias. Caso umas das verificações der negativo é retornada a matriz nula. Caso todas as verificações passem, uma nova matriz com as mesmas dimensões das de entrada é gerada. Em seguida as duas matrizes de entrada têm seus valores percorridos e somados, com o resultado sendo salvo na matriz nova. Por fim, é retornado a matriz gerada.

***Texto

Descrição gerada automaticamente***

Figura 43: Implementação do *matrix\_sum*

* ***matrix\_dif:***  Essa operação recebe como entrada duas matrizes na qual é verificada se possuem as mesmas dimensões e se elas não estão vazias. Caso umas das verificações der negativo é retornada a matriz nula. Caso todas as verificações passem, uma nova matriz com as mesmas dimensões das de entrada é gerada. Em seguida as duas matrizes de entrada têm seus valores percorridos e subtraídos (valor da Matriz da A - valor da Matriz B), com o resultado sendo salvo na matriz nova. Por fim, é retornado a matriz gerada.

***Texto

Descrição gerada automaticamente***

Figura 44: Implementação do *matrix\_dif*

* ***matrix\_compare:*** Essa operação recebe como entrada duas matrizes na qual é verificada se possuem as mesmas dimensões e se elas não estão vazias. Caso umas das verificações der negativo é retornado inteiro 0 (simboliza o *false*). Caso todas as verificações passem, as duas matrizes são percorridas verificando se os valores de ambas são iguais para a mesma posição. Caso algum valor seja diferente é retornado o inteiro 0, caso todos sejam iguais então é retornado o valor 1 (simboliza o *true*).

***Tela de computador com letras e números em fundo preto

Descrição gerada automaticamente***

Figura 45: Implementação do *matrix\_compare*

* ***matrix\_det:*** A função tem como objetivo encontrar o determinante da matriz passada como parâmetro da função. Para isso a matriz original é escalonada a fim de transformá-la em uma matriz triangular, uma vez que se torna triangular basta multiplicar a sua diagonal principal para encontrar o determinante. Sendo assim a função verifica se a matriz é quadrada e possui valores, caso não seja quadrada ou esteja vazia então é retornado o 0.

***Texto

Descrição gerada automaticamente***

Figura 46: Implementação do *matrix\_det*

* ***matrix\_transpose:*** A função recebe como entrada uma matriz na qual é verificada a existência de valores nela, caso não possua é retornada uma matriz nula. Caso passe pela verificação, uma nova matriz é gerada. As duas são percorridas, mas de maneiras diferentes, a matriz de entrada é percorrida de linha em linha enquanto a matriz criada é percorrida de coluna a coluna, assim o valor percorrido na matriz de entrada é copiado para a matriz gerada. Ao fim das iterações a matriz criada é retornada.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 47: Implementação do *matrix\_transpose*

* ***matrix\_inverse:*** A função tem como objetivo encontrar a matriz inversa para a matriz que é passada como parâmetro de entrada. Para isso é utilizada a decomposição em LU, na qual a matriz original é decomposta em uma série de outras matrizes montando um sistema com a matriz identidade a fim de achar os valores da matriz inversa. Esse método verifica se a matriz de entrada é quadrada e se ele contém algum valor, caso não seja quadrada ou esteja vazia é retornado uma matriz nula. Em seguida é gerado duas matrizes, uma sendo a cópia da matriz entrada e a outra sendo a matriz identidade. Ao final da decomposição o resultado desejado estará na matriz que originalmente era identidade sendo ela o retorno da função. Note que nesse método é selecionado um pivô a fim de decompor a matriz original em uma triangular, esse pivô é escolhido a partir de um valor contido na diagonal principal, dessa forma se algum valor for zero nessa diagonal da matriz entrada, a função retornará uma indeterminação.

***Texto

Descrição gerada automaticamente***

Figura 48: Implementação do *matrix\_inverse*

* ***print\_matrix:*** Função que recebe uma *string* e uma matriz e a percorre imprimindo cada valor contido nela.

***Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente***

Figura 49: Implementação do *print\_matrix*

1. **CONCLUSÃO**

A programação em tempo real se encontra em diversas situações do nosso cotidiano, dessa forma aprender a desenvolver esses sistemas é de grande importância para a evolução e distribuição deles. Para que isso seja possível o primeiro passo é a criação de um ambiente de desenvolvimento com as ferramentas necessárias para essa tarefa.

A configuração desse ambiente utilizando a máquina virtual WSL no sistema operacional Windows 10 para emular uma distribuição Linux se mostrou bastante simples e de fácil reprodução. Para este trabalho foram utilizadas 3 máquinas diferentes, um computador do Laboratório de Informática I (LIFT) da UFAM, um notebook e um desktop pessoal, e o processo de instalação do Ubuntu bem como o das bibliotecas para compilação em *C* (*build-essentials*) foram rápidos em todos os dispositivos.

O arquivo Makefile também fazem parte das ferramentas que agilizam o desenvolvimento nessa linguagem automatizando os comandos para a compilação e geração do executável. O editor de texto utilizado foi o *VSCode* com as extensões para a linguagem *C* e para acessar a máquina virtual com o Ubuntu. Outra ferramenta bastante útil ao se utilizar várias máquinas foi a plataforma do *Github* que permitia manter os arquivos do projeto sempre atualizados em todas as máquinas.

Por fim, as ADTs criadas serão de fundamental importância para o decorrer da disciplina. Foi possível implementar as ADTs de *string* dinâmica, de operações matriciais em ponto flutuante e integração de funções polinomiais. O processo de desenvolvimento deles permitiu revisar os conceitos de programação nessa linguagem bem como revisar os métodos matemáticos por trás desses cálculos. Dessa forma, o trabalho teve grande contribuição didática para o discente envolvido.

**5. REFERÊNCIAS**

CAVALCANTE, André. FTL066–Programação de Sistemas de Tempo Real, Laboratório 1 - Preparação do Ambiente. Manaus – UFAM, 2022.

FEOFILOFF, Paulo. Tipos abstratos de dados (ADTs). Departamento de Ciência da Computação, Instituto de Matemática e Estatística da USP, 2017. Disponível em: <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/adt.html#:~:text=Um%20ADT%20%C3%A9%20um%20conjunto,Exemplo%20de%20ADT%3A%20pilha>. Acesso em: 21/07/2022.

MAIN, Michael; SAVITCH, Walter; Tipos Abstratos de Dados. Traduzido de Maria Carolina Monard e José Augusto Baranauskas. The Benjamin/Cummings Publishing Company, 1995. Disponível em: <https://dcm.ffclrp.usp.br/~augusto/teaching/aedi/AED-I-OOP-ADT.pdf>. Acesso em: 21/07/2022.