



UNIVERSIDADE FEDERAL DE OURO PRETO

CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DO TRABALHO PRÁTICO

Integrante:

Yann Eduardo de Souza Silva

Ouro Preto

2024

Árvore Rubro-Negra

Introdução

Uma árvore rubro-negra, também conhecida como árvore vermelho-preto ou red-black, é um tipo de árvore binária de busca balanceada. Em contraste com as árvores AVL, que usam a altura das subárvores para o balanceamento, as árvores rubro-negras empregam um esquema de coloração dos nós para manter o balanceamento da árvore. Embora as árvores AVL e as árvores rubro-negras possuam a mesma complexidade computacional teórica para operações como inserção, remoção e busca ($O(\log N)$), na prática, elas diferem em termos de desempenho. A árvore AVL é mais rápida na operação de busca, enquanto a árvore rubro-negra é mais rápida nas operações de inserção e remoção.

A principal diferença entre as árvores AVL e as árvores rubro-negras reside no grau de rigidez do balanceamento. As árvores AVL têm um balanceamento mais rígido, o que pode resultar em um maior custo na operação de inserção e remoção. No entanto, as árvores rubro-negras apresentam um balanceamento menos rigoroso em comparação com as AVL, o que acelera a operação de busca.

No geral, a escolha entre árvores AVL e árvores rubro-negras depende das necessidades específicas de cada aplicação. Se a operação de busca é a mais comum, a árvore AVL pode ser mais adequada devido à sua eficiência nessa operação. Por outro lado, se as inserções e remoções são mais frequentes, a árvore rubro-negra pode ser uma escolha melhor devido à sua menor rigidez no balanceamento.

Para balancear, a árvore rubro-negra contém nós com um bit extra de armazenamento para indicar sua cor, que pode ser vermelho ou preto. Ao restringir como os nós podem ser coloridos em qualquer caminho da raiz até uma folha, as árvores rubro-negras garantem que nenhum desses caminhos seja mais longo que o dobro do comprimento de qualquer outro, resultando em uma árvore aproximadamente balanceada.

Cada nó da árvore foi representada pela estrutura RBtree, onde é composta pelos campos:

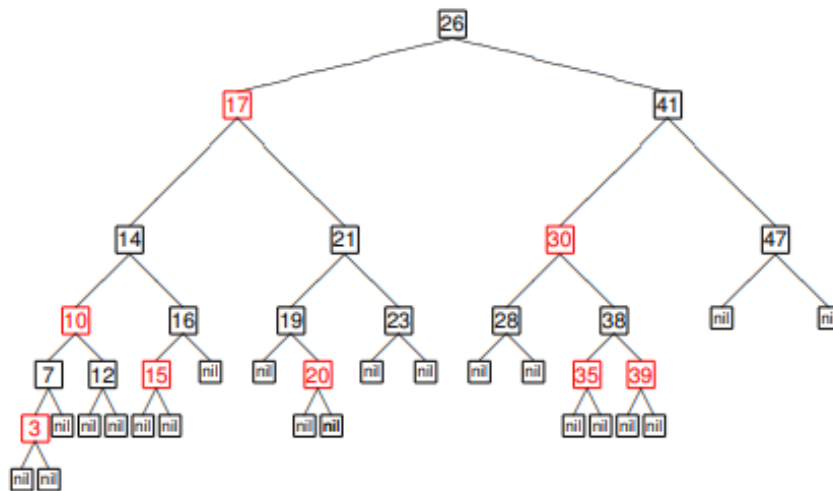
- Cor (1 bit): pode ser vermelho ou preto.
- Item: indica o que vai ser armazenado dentro do nó.
- Esquerda,direita: ponteiros que apontam para a subárvore esquerda e direita.
- Pai: ponteiro que aponta para o nó pai. O campo pai do nó raiz aponta para NULL

Uma árvore rubro-negra é uma árvore binária de busca, com algumas propriedades adicionais, que são:

- Todo nó é vermelho ou preto.
- A raiz é preta.
- Toda folha(NULL) é preta.
- Se um nó é vermelho, então ambos os seus filhos são pretos.
- Para cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o mesmo número de nós pretos.

A Figura 1 ilustra um exemplo de árvore rubro-negra, destacando como todos os nós em uma árvore rubro-negra são vermelhos ou pretos, os filhos de um nó vermelho são ambos pretos e como todos os caminhos simples desde um nó até uma folha descendente contêm o mesmo número de nós pretos. Cada folha, representada como NIL, é preta.

Figura 1: Exemplo de árvore rubro-negra



Autoria: Siang Wun Song - Universidade de São Paulo - IME/USP

Implementação

O código desenvolvido contém um total de 11 funções, tendo que duas delas são as funções principais “insercao” e “balanceamento”, e o restante são funções auxiliares.

A estratégia utilizada neste código para implementar uma árvore rubro-negra envolveu as seguintes etapas:

1. **Definição de Estruturas e Constantes:** Foram definidas estruturas para representar os itens da árvore e os nós da própria árvore, além de constantes para representar as cores dos nós. Elas estão demonstrados abaixo:
 - a. A estrutura `TItem` é definida para representar os elementos da árvore, contendo um inteiro (`idade`) e uma string (`nome`).
 - b. A estrutura `RBtree` é definida para representar os nós da árvore rubro-negra. Ela contém um elemento do tipo `TItem`, além de ponteiros para os nós esquerdo (`pEsq`) e direito (`pDir`), um ponteiro para o nó pai (`pai`) e um inteiro para representar a cor do nó (`cor`).
 - c. A estrutura `Tree` é definida para representar a própria árvore, contendo apenas um ponteiro para a raiz da árvore.
 - d. São definidas constantes `RED` e `BLACK` para representar as cores dos nós da árvore.
2. **Manipulação de Nós e Árvore:** Funções foram criadas para inicializar uma árvore, alocar memória para a árvore e seus nós, desalocar a memória alocada e criar um novo nó da árvore. Elas estão demonstradas abaixo:
 - a. (1ª função) `inicia`: Inicializa uma árvore, definindo sua raiz como `NULL`.
 - b. (2ª função) `criaNo`: Aloca memória para um novo nó da árvore, inicializando-o com a cor vermelha.
 - c. (3ª função) `alocarArvore`: Aloca memória para uma nova árvore e a inicializa.
 - d. (4ª função) `desalocarArvore`: Desaloca a memória ocupada por uma

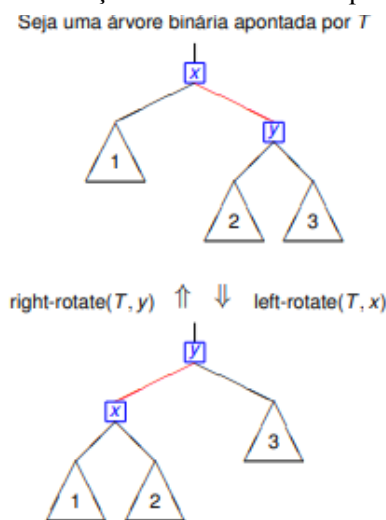
árvore e seus nós.


- e. (5ª função) `desalocarNO`: Desaloca a memória ocupada por um nó e seus descendentes.

- 3. **Rotações:** A operação de inserção em uma árvore rubro-negra com n chaves leva tempo $O(\lg n)$. No entanto, essa operação pode modificar a estrutura da árvore, potencialmente violando suas propriedades. Para corrigir essas violações, precisamos ajustar as cores dos nós na árvore e também reorganizar os ponteiros.

As reorganizações dos ponteiros são realizadas por meio de rotações, uma operação local em uma árvore de busca binária que mantém sua propriedade. As rotações têm complexidade constante $O(1)$. Existem dois tipos de rotações: rotação à esquerda e rotação à direita. Ao executar uma rotação à esquerda em um nó x , assumimos que seu filho da direita y não é NULL. Durante a rotação à esquerda, ocorre uma mudança na estrutura da subárvore, tornando y a nova raiz da subárvore. Nesse processo, x se torna o filho da esquerda de y , e o filho da esquerda de y se torna o filho da direita de x . A figura 2 mostra como ocorre a rotação da esquerda e a rotação da direita.

Figura 2: rotação da direita e da esquerda



Autoria:  Siang Wun Song - Universidade de São Paulo - IME/USP

Os protótipos `rotacaoEsquerda` e `rotacaoDireita` que implementam as rotações esquerda e direita, respectivamente, necessárias para o balanceamento da árvore. As rotações acontecem da seguinte forma:

- a. (6ª função) `rotacaoEsquerda`: Durante a rotação à esquerda, um nó é deslocado para baixo e para a esquerda, enquanto seu filho direito é promovido para ocupar sua posição anterior. Isso acontece quando o filho direito de um nó é elevado para substituir seu pai. O nó rotacionado passa a ser o filho esquerdo do seu antigo filho direito. Essa rotação é usada para reequilibrar a árvore quando ocorre uma inserção ou remoção à direita do pai, causando um desequilíbrio.
- b. (7ª função) `rotacaoDireita`: Na rotação à direita, um nó é deslocado para baixo e para a direita, enquanto seu filho esquerdo é promovido para ocupar

sua posição anterior. Isso ocorre quando o filho esquerdo de um nó é promovido para substituir seu pai. O nó rotacionado passa a ser o filho direito do seu antigo filho esquerdo. Essa rotação é usada para reequilibrar a árvore quando ocorre uma inserção ou remoção à esquerda do pai, causando um desequilíbrio.

Na Figura 3, é apresentado o pseudocódigo utilizado para implementar tanto a rotação esquerda quanto à rotação direita. Dado que essas operações são simétricas, apenas esse pseudocódigo foi utilizado para ambas as rotações.

Figura 3: pseudocódigo das rotações

```

LEFT-ROTATE(T, x)
1  y = x.right           // set y
2  x.right = y.left       // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil
4      y.left.p = x
5  y.p = x.p             // link x's parent to y
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x           // put x on y's left
12 x.p = y

```

Autoria: Introduction to Algorithms, Third Edition, Thomas H. Cormen

4. **Inserção:** A inserção de um nó em uma árvore rubro-negra de n nós pode ser realizada no tempo $O(\lg n)$. Usamos uma versão ligeiramente modificada do algoritmo de inserção de uma árvore binária, para inserir o nó “pNo” na árvore “arvore” como se ela fosse uma árvore de pesquisa binária comum, e depois colorimos “pNo” de vermelho. Para garantir que as propriedades vermelho-preto sejam preservadas, chamamos então um procedimento auxiliar, que é o balanceamento, para recolorir os nós e executar rotações. A chamada da função “(8°função)insercao(Tree *arvore, RBtree *pNo)” insere o nó “pNo” na árvore rubro-negra.

Na Figura 4, é apresentado o pseudocódigo utilizado para implementar a inserção.

Figura 4: pseudocódigo da inserção

```

RB-INSERT(T, z)
1  y = T.nil
2  x = T.root
3  while x ≠ T.nil
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == T.nil
10     T.root = z
11 elseif z.key < y.key
12     y.left = z
13 else y.right = z
14 z.left = T.nil
15 z.right = T.nil
16 z.color = RED
17 RB-INSERT-FIXUP(T, z)

```

Autoria: Introduction to Algorithms, Third Edition, Thomas H. Cormen

5. **Balanceamento:** O balanceamento das árvores rubro-negras é realizado por meio de rotações e ajustes de cores a cada inserção ou remoção. Essas operações mantêm o equilíbrio da árvore e corrigem possíveis violações de suas propriedades. Apesar das operações de balanceamento, o custo máximo de qualquer algoritmo em uma árvore rubro-negra é $O(\log N)$, o que garante eficiência em operações mesmo em árvores grandes.

Essa função é responsável por garantir que as propriedades da árvore rubro-negra sejam preservadas após a inserção de um novo nó (p_{No}). Ela percorre a árvore a partir do nó inserido até a raiz, realizando operações de recoloração e rotações conforme necessário para manter a integridade da árvore.

Os quatro casos tratados na função correspondem a diferentes configurações que podem surgir após a inserção de um nó vermelho (p_{No}) na árvore. Aqui está uma explicação detalhada de cada caso:

- a. **Caso 1: Tio Vermelho:** Se o tio de p_{No} (armazenado em p_{Aux}) é vermelho, isso indica uma violação das propriedades da árvore rubro-negra. Nesse caso, realizamos uma recoloração: o pai de p_{No} e o tio p_{Aux} são coloridos de preto, enquanto o avô de p_{No} é colorido de vermelho. Em seguida, movemos o nó p_{No} para o avô de p_{No} , pois o avô de p_{No} pode agora violar as propriedades da árvore. Esse caso é uma recoloração direta e não requer rotações.
- b. **Caso 2: Tio Preto e p_{No} é Filho Direito:** Se o tio p_{Aux} de p_{No} é preto e p_{No} é o filho direito de seu pai, isso forma uma configuração triangular. Nesse caso, realizamos uma rotação à esquerda em torno do pai de p_{No} para tornar a situação semelhante ao Caso 3, onde p_{No} é o filho esquerdo de seu pai.
- c. **Caso 3: Tio Preto e p_{No} é Filho Esquerdo:** Se o tio p_{Aux} de p_{No} é preto e p_{No} é o filho esquerdo de seu pai, isso forma uma configuração linear. Nesse caso, realizamos uma rotação à direita em torno do avô de p_{No} para reequilibrar a árvore. Após a rotação, recolorimos o pai de p_{No} para preto e o avô de p_{No} para vermelho.
- d. **Recoloração da Raiz:** No final do loop de balanceamento, garantimos que a raiz da árvore seja sempre preta, conforme exigido pelas propriedades da árvore rubro-negra.

Na Figura 5, é apresentado o pseudocódigo utilizado para implementar o balanceamento.

Figura 5: pseudocódigo do balanceamento

```

RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK           // case 1
6              y.color = BLACK           // case 1
7              z.p.p.color = RED         // case 1
8              z = z.p.p                 // case 1
9          else if z == z.p.right
10             z = z.p                   // case 2
11             LEFT-ROTATE(T, z)         // case 2
12             z.p.color = BLACK         // case 3
13             z.p.p.color = RED         // case 3
14             RIGHT-ROTATE(T, z.p.p)    // case 3
15         else (same as then clause
              with "right" and "left" exchanged)
16  T.root.color = BLACK

```

Autoria: Introduction to Algorithms, Third Edition, Thomas H. Cormen

6. **Leitura e Impressão:** Funções foram fornecidas para ler dados da entrada padrão e inseri-los na árvore, bem como para imprimir os nós da árvore em ordem crescente de idade. Os protótipos das funções são:
 - a. (10ª função) `printInOrder`: Imprime os nós da árvore em ordem crescente de idade.
 - b. (11ª função) `leArvore`: permite a inserção de elementos na árvore a partir da entrada padrão, lendo o nome e a idade de cada nó e chamando `insercao` para adicioná-lo à árvore.

Testes

Cinco testes que estão armazenados na pasta "tests" foram executados. Cada teste começa com uma linha que indica qual operação será realizada: : 1 - Inserção, 2 - Impressão InOrder da árvore, 0 - para sair do loop. A segunda linha depende da opção selecionada na primeira, caso seja 1, então o próximo dado é a quantidade de dados a serem inseridos. Após a quantidade de dados, são inseridos os dados na seguinte ordem: nome e idade. Para cada teste, o resultado esperado é os nomes ordenados de forma crescente.

Assim sendo, abaixo estará listado as entradas dos cinco testes feitos e suas respectivas saídas esperadas:

Teste 1 (entrada):

```

1
2
Luan
10
Ana

```

24

1

3

Lucas

23

Luisa

29

Laura

13

0

Saída:

Dados inOrder:

Nome: Luan

Idade: 10

Nome: Laura

Idade: 13

Nome: Lucas

Idade: 23

Nome: Ana

Idade: 24

Nome: Luisa

Idade: 29

Teste 2 (entrada):

1

4

Lucas

22

Luisa

21

Beatriz

32

Bruna

25

2

1

3

Lais

20

Maria

54

Lorena

19

0

Saída:

Dados inOrder:

Nome: Luisa

Idade: 21

Nome: Lucas

Idade: 22

Nome: Bruna

Idade: 25

Nome: Beatriz

Idade: 32

Dados inOrder:

Nome: Lorena

Idade: 19

Nome: Lais

Idade: 20

Nome: Luisa

Idade: 21

Nome: Lucas

Idade: 22

Nome: Bruna

Idade: 25

Nome: Beatriz

Idade: 32

Nome: Maria

Idade: 54

Teste 3 (entrada):

1

2

Barbara

24

Vitor

23

2

1

3

Karol

18

Bernardo

12

Marlon

28

0

Saída:

Dados inOrder:

Nome: Vitor

Idade: 23

Nome: Barbara

Idade: 24

Dados inOrder:

Nome: Bernardo

Idade: 12

Nome: Karol

Idade: 18

Nome: Vitor

Idade: 23

Nome: Barbara

Idade: 24

Nome: Marlon

Idade: 28

Teste 4 (entrada):

1

6

Maria

58

Mauro

63

Kamila

29

Pedro

30

Alana

38

Luisa

19

0

Saída:

Dados inOrder:

Nome: Luisa

Idade: 19

Nome: Kamila

Idade: 29

Nome: Pedro

Idade: 30

Nome: Alana

Idade: 38

Nome: Maria

Idade: 58

Nome: Mauro

Idade: 63

Teste 5 (entrada):

1

7

Marlene

42

Mariana

29

Luana

20

Vitoria

25

Liana

38

Elisa

16

Marcos

18

2

1

3

Mauricio

70

Lucas

23

Lauro

48

0

Saída:

Dados inOrder:

Nome: Elisa

Idade: 16

Nome: Marcos

Idade: 18

Nome: Luana

Idade: 20

Nome: Vitoria

Idade: 25

Nome: Mariana

Idade: 29

Nome: Liana

Idade: 38

Nome: Marlene

Idade: 42

Dados inOrder:

Nome: Elisa

Idade: 16

Nome: Marcos

Idade: 18

Nome: Luana

Idade: 20

Nome: Lucas

Idade: 23

Nome: Vitoria

Idade: 25

Nome: Mariana

Idade: 29

Nome: Liana

Idade: 38

Nome: Marlene

Idade: 42

Nome: Lauro

Idade: 48

Nome: Mauricio

Idade: 70

Analise

O programa foi analisado usando a ferramenta valgrind, a fim de identificar e verificar possíveis problemas de vazamento de memória. As imagens abaixo mostram os resultados das análises feitas pela ferramenta, apontam que não foram identificados vazamentos de memória nas alocações, conforme indicado nos textos gerados. A última imagem apresenta o resultado dos testes feitos pelo corretor de testes.

Saída do Teste 1:

```
● yann1106@yann-2005:~/tped1/tp3/tp$ valgrind --leak-check=full -s ./exe < 1.in
==65564== Memcheck, a memory error detector
==65564== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==65564== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==65564== Command: ./exe
==65564==
Dados inOrder:
Nome: Luan
Idade: 10
Nome: Laura
Idade: 13
Nome: Lucas
Idade: 23
Nome: Ana
Idade: 24
Nome: Luisa
Idade: 29
==65564==
==65564== HEAP SUMMARY:
==65564==   in use at exit: 0 bytes in 0 blocks
==65564==   total heap usage: 8 allocs, 8 frees, 5,408 bytes allocated
==65564==
==65564== All heap blocks were freed -- no leaks are possible
==65564==
==65564== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Tempo de execução do Teste 1:

```
yann1106@yann-2005:~/tped1/tp3/tp$ time ./exe < tests/1.in
Dados inOrder:
Nome: Luan
Idade: 10
Nome: Laura
Idade: 13
Nome: Lucas
Idade: 23
Nome: Ana
Idade: 24
Nome: Luisa
Idade: 29

real    0m0,007s
user    0m0,004s
sys     0m0,003s
```

Saída do Teste 2:

```
==65708== Memcheck, a memory error detector
==65708== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==65708== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==65708== Command: ./exe
==65708==
Dados inOrder:
Nome: Luisa
Idade: 21
Nome: Lucas
Idade: 22
Nome: Bruna
Idade: 25
Nome: Beatriz
Idade: 32
Dados inOrder:
Nome: Lorena
Idade: 19
Nome: Lais
Idade: 20
Nome: Luisa
Idade: 21
Nome: Lucas
Idade: 22
Nome: Bruna
Idade: 25
Nome: Beatriz
Idade: 32
Nome: Maria
Idade: 54
==65708==
==65708== HEAP SUMMARY:
==65708==      in use at exit: 0 bytes in 0 blocks
==65708==    total heap usage: 10 allocs, 10 frees, 5,520 bytes allocated
==65708==
==65708== All heap blocks were freed -- no leaks are possible
==65708==
==65708== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Tempo de execução do Teste 2:

```
yann1106@yann-2005:~/tped1/tp3/tp$ time ./exe < tests/2.in
Dados inOrder:
Nome: Luisa
Idade: 21
Nome: Lucas
Idade: 22
Nome: Bruna
Idade: 25
Nome: Beatriz
Idade: 32
Dados inOrder:
Nome: Lorena
Idade: 19
Nome: Lais
Idade: 20
Nome: Luisa
Idade: 21
Nome: Lucas
Idade: 22
Nome: Bruna
Idade: 25
Nome: Beatriz
Idade: 32
Nome: Maria
Idade: 54

real    0m0,007s
user    0m0,003s
sys     0m0,005s
```

Saída do Teste 3:

```
yann1106@yann-2005:~/tped1/tp3/tp$ valgrind --leak-check=full -s ./exe < 3.in
==4722== Memcheck, a memory error detector
==4722== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4722== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4722== Command: ./exe
==4722==
Dados inOrder:
Nome: Vitor
Idade: 23
Nome: Barbara
Idade: 24
Dados inOrder:
Nome: Bernardo
Idade: 12
Nome: Karol
Idade: 18
Nome: Vitor
Idade: 23
Nome: Barbara
Idade: 24
Nome: Marlon
Idade: 28
==4722==
==4722== HEAP SUMMARY:
==4722==   in use at exit: 0 bytes in 0 blocks
==4722== total heap usage: 8 allocs, 8 frees, 5,408 bytes allocated
==4722==
==4722== All heap blocks were freed -- no leaks are possible
==4722==
==4722== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Tempo de execução do Teste 3:

```
yann1106@yann-2005:~/tped1/tp3/tp$ time ./exe < tests/3.in
Dados inOrder:
Nome: Vitor
Idade: 23
Nome: Barbara
Idade: 24
Dados inOrder:
Nome: Bernardo
Idade: 12
Nome: Karol
Idade: 18
Nome: Vitor
Idade: 23
Nome: Barbara
Idade: 24
Nome: Marlon
Idade: 28

real    0m0,005s
user    0m0,004s
sys     0m0,000s
```

Saída 4:

```
yann1106@yann-2005:~/tped1/tp3/tp$ valgrind --leak-check=full -s ./exe < 4.in
==4863== Memcheck, a memory error detector
==4863== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4863== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4863== Command: ./exe
==4863==
Dados inOrder:
Nome: Luisa
Idade: 19
Nome: Kamila
Idade: 29
Nome: Pedro
Idade: 30
Nome: Alana
Idade: 38
Nome: Maria
Idade: 58
Nome: Mauro
Idade: 63
==4863==
==4863== HEAP SUMMARY:
==4863==    in use at exit: 0 bytes in 0 blocks
==4863==   total heap usage: 9 allocs, 9 frees, 5,464 bytes allocated
==4863==
==4863== All heap blocks were freed -- no leaks are possible
==4863==
==4863== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Tempo de execução do Teste 4:

```
yann1106@yann-2005:~/tped1/tp3/tp$ time ./exe < tests/4.in
Dados inOrder:
Nome: Luisa
Idade: 19
Nome: Kamila
Idade: 29
Nome: Pedro
Idade: 30
Nome: Alana
Idade: 38
Nome: Maria
Idade: 58
Nome: Mauro
Idade: 63

real    0m0,009s
user    0m0,008s
sys     0m0,001s
```

Saída 5:

```
yann1106@yann-2005:~/tped1/tp3/tp$ valgrind --leak-check=full -s ./exe < 5.in
==4917== Memcheck, a memory error detector
==4917== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4917== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4917== Command: ./exe
==4917==
Dados inOrder:
Nome: Elisa
Idade: 16
Nome: Marcos
Idade: 18
Nome: Luana
Idade: 20
Nome: Vitoria
Idade: 25
Nome: Mariana
Idade: 29
Nome: Liana
Idade: 38
Nome: Marlene
Idade: 42
Dados inOrder:
Nome: Elisa
Idade: 16
Nome: Marcos
Idade: 18
Nome: Luana
Idade: 20
Nome: Lucas
Idade: 23
Nome: Vitoria
```



```

Nome: Vitoria
Idade: 25
Nome: Mariana
Idade: 29
Nome: Liana
Idade: 38
Nome: Marlene
Idade: 42
Nome: Lauro
Idade: 48
Nome: Mauricio
Idade: 70
==4917==
==4917== HEAP SUMMARY:
==4917==   in use at exit: 0 bytes in 0 blocks
==4917== total heap usage: 13 allocs, 13 frees, 5,688 bytes allocated
==4917==
==4917== All heap blocks were freed -- no leaks are possible
==4917==
==4917== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Tempo de execução do Teste 5:

```

Nome: Luana
Idade: 20
Nome: Lucas
Idade: 23
Nome: Vitoria
Idade: 25
Nome: Mariana
Idade: 29
Nome: Liana
Idade: 38
Nome: Marlene
Idade: 42
Nome: Lauro
Idade: 48
Nome: Mauricio
Idade: 70

real    0m0,007s
user    0m0,004s
sys     0m0,004s

```

Saída do Corretor:

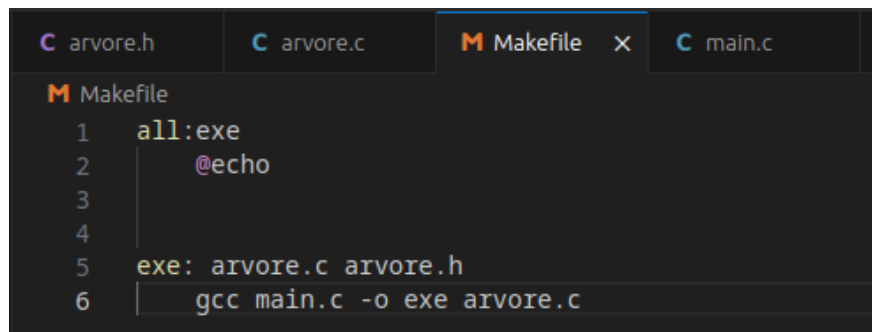
```

yann1106@yann-2005:~/tped1/tp3/tp$ make

yann1106@yann-2005:~/tped1/tp3/tp$ python3 corretor.py
Analisando atividade:
  1.in OK
  2.in OK
  3.in OK
  4.in OK
  5.in OK
Nota na atividade: 10.00
yann1106@yann-2005:~/tped1/tp3/tp$

```

Make:



```
1 all:exe
2     @echo
3
4
5 exe: arvore.c arvore.h
6     gcc main.c -o exe arvore.c
```

Conclusão

Este estudo abordou a implementação de uma árvore rubro-negra em linguagem C, abrangendo operações de inserção e balanceamento. Os principais desafios enfrentados incluíram compreender detalhadamente as propriedades da árvore rubro-negra e os algoritmos associados, além de garantir a correta implementação das rotações e do balanceamento para manter a integridade da árvore.

A implementação abordou a criação das estruturas de dados necessárias, como nós da árvore e a própria árvore, juntamente com as operações de inserção e balanceamento. As rotações esquerda e direita foram essenciais para o processo de balanceamento, garantindo que a árvore mantivesse suas propriedades.

Em resumo, o estudo proporcionou uma compreensão prática das árvores rubro-negras, desde os conceitos fundamentais até a implementação das operações básicas, destacando os desafios enfrentados durante o processo.

Referência bibliográfica

<https://www.ime.usp.br/~song/mac5710/slides/08rb.pdf>

<https://docente.ifrn.edu.br/robinsonalves/disciplinas/estruturas-de-dados/ArvRN.pdf>

<https://www.inf.ufpr.br/andre/textos-CI1165/Introduction%20to%20Algorithms%20-%203rd%20Edition.pdf>