Segundo Trabalho Avaliativo de Estrutura de Dados Básica II

Raoni Silva, Pedro Galvão, Hélio Lima e Thiago Freire

January 9, 2025

Turma 35M34
Unidade 2

Contents

1 Ambiente Computacional

Todas as implementações foram executadas e cronometradas no mesmo ambiente computacional, para uma comparação justa e adequada entre as diferentes formas de implementação das funções citadas no roteiro avaliativo.

O ambiente computacional consiste em uma máquina virtual no servidor pessoal de um dos integrantes do grupo. A máquina virtual possui dois núcleos do processador, 2GB de memória RAM e 32GB de disco rígido à sua disposição, rodando uma distribuição Linux conhecida como Arch Linux, com uma insta- lação mínima, contendo somente os serviços necessários para funcionamento do sistema operacional e o serviço de servidor SSH, para conexão remota com a máquina.

Todas as implementações foram cronometradas utilizando os métodos apropriados para cada linguagem de programação e seus respectivos tempos de execução foram armazenados em um arquivo final, para serem analisados no presente relatório.

As implementações da heap foram feitas em Rust, e as árvores foram implementadas em C. Até existiu a tentativa de criar a árvore rubro-negra em rust, mas devido a necessidade de ter referências cíclicas, a implementação em Rust fica muito complicada.

2 Heap

Nessa seção, são explicadas as estruturas de MinHeap e MaxHeap. Ambas foram implementadas em Rust, devido as atividades da primeira unidade, pois as ordenações da primeira únidade foram feitas em Rust. Primeiramente explicaremos a Max Heap e após isso, mais resumidamente devido a semelhança, a Min Heap.

2.1 Max Heap

A Max Heap é uma espécie de árvore, que é comumente implementada como lista. A única condição para que uma árvore/lista seja considerada uma Max Heap, é que para todo nó, seu filhos devem ter prioridade menor que o pai.

Desse modo, nota-se que o maior elemento da Max Heap sempre será a raiz dela (ou o H[0], no caso da lista).

2.1.1 Estrutura

Na implementação da Max Heap, criamos um WrapperType(uma classe), para encapsular o funcionamento da Max Heap:

```
pub struct MaxHeap<T> {
    data: Vec<T>,
}
```

2.1.2 Alteração de prioridade

Para realizar a alteração de prioridade na heap(o tira casaco, bota casaco dela), é preciso implementar as funções de subir e descer na heap. Elas servem para manter a principal propriedade da (max)heap: cada nó tem prioridade maior que seus filhos.

1. Função subir

Para a função de subir(bubble_up), a implementação é simples. Pegamos a heap(&mut self) e a posição que ira subir como argumentos. Devido as propriedades da heap, sabemos que o pai da $self[i]$ está na posição $i/2$, e dessa forma verificamos se o filho tem prioridade maior que o pai. Se for o caso, as posições do filho e do pai são trocadas, e então chama-se a função recursivamente na posição do pai.

```
pub fn bubble_up(&mut self, mut index: usize) {
    while index > 0 {
        let parent = (index - 1) / 2;
        if self[index] <= self[parent] {
            break;
        }
        self.swap(index, parent);
        index = parent;
    }
}
```

2. Função descer

Para a função de descer, é um pouco mais complicado. Visto que cada item da heap terá 2 filhos, é preciso levar em conta os dois, para decidir o que fazer no algoritmo.

Primeiramente, pegamos a quantidade de elementos na Heap e então fazemos uma iteração.

Para cada iteração, comparamos a prioridade do index atual com a prioridade de seus filhos, caso algum dos filhos seja maior que o pai, realizamos o swap do pai com o filho, e repetimos o processo. Se nenhum dos filhos é maior que o pai, significa que o item desceu até a posição correta, e paramos o loop.

```
pub fn bubble_down(&mut self, mut index: usize) {
    let last_index = match self.len() {
        0 => 0,
        n => n - 1,
    };
    loop {
        let left_child = (2 * index) + 1;
        let right_child = (2 * index) + 2;
        let mut largest = index;
        if left_child <= last_index && self[left_child] >
           self[largest] {
            largest = left_child;
        }
        if right_child <= last_index && self[right_child] >
           self[largest] {
            largest = right_child;
        }
        if largest == index {
            break;
        }
        self.swap(index, largest);
        index = largest;
    }
}
```

2.1.3 Inserção

Adiciona-se o valor ao final da lista e então usa a função subir(bubble_up) para corrigir a prioridade do novo valor inserido.

```
pub fn push(&mut self, value: T) {
    self.data.push(value);
    self.bubble_up(self.data.len() - 1);
}
```

2.1.4 Remoção

Se a lista está vazia, apenas retorna *None*.

Se não, troca-se a posição do último elemento com o primeiro, retira-se o novo último da lista e utiliza-se a função descer(bubble_down) no novo primeiro elemento.

```
pub fn pop(&mut self) -> Option<T> {
    if self.is_empty() {
        return None;
    }

    let last_index = self.len() - 1;
    self.swap(0, last_index);
    let max_value = self.data.pop();
    self.bubble_down(0);
    max_value
}
```

2.1.5 Construção da MaxHeap

Para a construção de uma MaxHeap a partir de uma lista, foi utilizada a trait (interface/typeclass) From<Vec<T>>. Essa trait é o método padrão para fazer "mapeamento de tipos", ou seja, mapear um tipo A em um tipo B. Nesse caso, mapeamos o tipo Vec<T> em MaxHeap<T>.

Quanto ao algoritmo, ele se baseia em considerar que as folhas da heap já são heaps. Com isso, temos que a partir da posição $len/2 + 1$, a lista já é uma heap.

Desse modo, só o que falta é organizar a MaxHeap de 0 até $len/2$. Para isso, usamos a função descer(bubble_down) de $len/2$ até 0 e então retornamos a heap.

```
impl<T: Default + Ord> From<Vec<T>> for MaxHeap<T> {
    fn from(data: Vec<T>) -> Self {
        let mut heap = MaxHeap { data };
        let len = heap.len();
        for i in (0..len / 2).rev() {
            heap.bubble_down(i);
        }
        heap
    }
}
```

2.1.6 (Max)Heapsort

No heapsort, o que fazemos é consumir a MaxHeap para retornar um Vetor ordenado.

Para isso, criamos um vetor mutável com o tamanho da MaxHeap, e como o primeiro elemento da MaxHeap é sempre o maior dela, se você retirar repetidamente os valores da heap e inseri-los na lista, você terá uma lista ordenada descendente. Quando a MaxHeap é esvaziada, temos uma lista ordenada "ao contrário", por isso usamos a função reverse retornar uma lista ordenada ascendente.

```
pub fn heapsort(mut self) -> Vec<T> {
    let mut sorted = Vec::with_capacity(self.len());
```

```
    while let Some(max) = self.pop() { sorted.push(max); }
    sorted.reverse();
    sorted
}
```

2.2 Min Heap

Em uma Min Heap, a regra a ser seguida é o contrário da Max Heap: Para todo nó, o os filhos dele devem ter prioridade MAIOR que ele, ou seja, a prioridade do pai é sempre MENOR que seus filhos. Desse modo, a implementação da Min Heap é muito similar, a diferença é que é necessário "flipar" as condicionais nas funções de subir e descer.

Além disso, no (min)heapsort, não será necessário fazer um .reverse() na lista, pois os elementos já estarão em ordem crescente devido a propriedade do Min Heap.

2.3 Comparação de Heapsort

O heapsort foi feito de acordo com a linguagem utilizada na primeira unidade, o rust. Desse modo, foi utilizado o mesmo código da primeira unidade, apenas adicionando no código o comando para que ele rode também o heapsort.

Vale ressaltar que os resultados dessa vez foram muito mais expressivos nas listas maiores, visto que foi pedida a ordenação de listas com 1 milhão de itens.

Dito isso, os novos resultados mostram que existe pouca diferença entre os algoritmos mais rápidos, com exceção do quicksort, que acabou sofrendo mais. O motivo disso é que a sua complexidade no pior caso é $O(n^2)$, e com valores muito grandes, esse fator começa a pesar mais. Comentando sobre o heapsort, percebe-se que ele tem um comportamento muito similar ao mergesort, e isso se deve ao fato que ambos são algoritmos estáveis, no sentido de que ambos sempre realizam $O(n * logn)$ operações para ordenar um vetor.

Tamanho	Heap(Min)	Heap(Max)	Merge(R)	Merge(I)	Quick(R)	Quick(I)
10.000	1.76ms	1.91ms	5.17ms	4.95ms	1.7ms	1.18ms
100.000	21.76ms	22.23ms	48.76ms	40.11ms	53.96ms	21.28ms
1.000.000	491.96ms	509.67ms	475.00ms	451.86ms	4.07s	1.14s

Table 1: Comparação de algoritmos $O(n * log(n))$

Agora, comparando o heapsort com o bubblesort, temos uma situação engraçada, com 1 milhão de itens, o bubble sort demorou quase duas horas para realizar a ordenação, enquanto o heapsort demorou meio segundo. Para ter noção da discrepância, com alguns cálculos e alguns arredondamentos, chegamos a conclusão de que enquanto o bubblesort ordena um array de um milhão de elementos, o heapsort consegue ordenar cerca de 13 mil arrays com 1 milhão de elementos.

Isso demonstra muito evidentemente(foram mais de 3 horas esperando isso rodar) que assintoticamente, o algoritmo bubble sort é péssimo.

No mais, o heapsort é uma opção bem sólida, que tem uma performance similar ao mergesort, mas com uma maior eficiência de espaço.

Tamanho	Heap(Min)	Heap(Max)	Bubble(R)	Bubble(I)
10.000	1.76ms	1.91ms	345.98ms	244.47ms
100.000	21.76ms	22.23ms	34.46s	34.50s
1.000.000	491.96ms	509.67ms	1h44m	1h51m

Table 2: Comparação Heapsort e Bubblesort

3 Árvore Binária

A implementação da estrutura de árvore binária foi feita em C devido ao controle preciso de memória e à utilização eficiente de ponteiros, características essenciais dessa linguagem.

3.1 Definição

Uma árvore binária é uma estrutura de dados composta por um conjunto finito de elementos, chamados de nós, sendo que o primeiro nó, denominado raiz, é o ponto inicial da árvore e o os nós da base são conhecidos como folhas. Em uma árvore binária, cada nó pode ter no máximo dois filhos, um à esquerda e outro à direita.

A árvore binária tem a propriedade de que todos os nós de uma subárvore à direita de um nó são maiores do que o valor armazenado na raiz desse nó, enquanto todos os nós de uma subárvore à esquerda de um nó são menores do que o valor armazenado na raiz desse nó. Além disso, cada subárvore, formada pelos filhos à esquerda e à direita de qualquer nó, é também uma árvore binária por si mesma. Isso torna a estrutura recursiva, em que cada nó pode ser considerado a raiz de uma nova árvore binária.

Essa organização permite operações eficientes de busca, inserção e remoção, com a garantia de que a árvore estará estruturada de forma hierárquica e balanceada.

3.2 Estrutura

Na implementação da árvore binária, utilizamos uma estrutura do tipo struct para representar os nós da árvore. Cada nó contém três componentes principais: um valor, representado pela chave, e dois ponteiros, um para a subárvore à esquerda e outro para a subárvore à direita. A seguir, temos a definição da estrutura:

```
1 typedef struct arvore_t {
2   int chave;
3   struct arvore_t *esq;
4   struct arvore_t *dir;
5 } arvore_t;
```

3.3 Operações de Manipulação

3.3.1 Inserção

Na operação de inserção em uma árvore binária, é fundamental que as propriedades da árvore sejam preservadas. Isso significa que todos os nós da subárvore à esquerda de um nó devem conter valores menores que a chave desse nó, e todos os nós da subárvore à direita devem conter valores maiores. Além disso, todo novo nó inserido na árvore sempre será uma folha, ou seja, não possuirá filhos imediatamente após sua criação.

A seguir, apresentamos o código que implementa essa operação:

```c
arvore_t *inserir(arvore_t *arvore, int chave) {
  if (arvore == NULL) {
    arvore = (arvore_t *)malloc(sizeof(arvore_t));
    arvore->chave = chave;
    arvore->esq = NULL;
    arvore->dir = NULL;
  } else if (chave < arvore->chave) {
    arvore->esq = inserir(arvore->esq, chave);
  } else if (chave > arvore->chave) {
    arvore->dir = inserir(arvore->dir, chave);
  }
  return arvore;
}
```

3.3.2 Criação de Árvore a partir de uma Lista

Para construir uma árvore binária a partir de uma lista, de forma que ela fique balanceada ou aproximadamente balanceada, foi adotada a seguinte estratégia:

Etapa 1: Início.

Etapa 2: Verifica se o tamanho da lista é 0.

- Se sim, retorna `NULL`.
- Caso contrário, chama a função `construir_arvore`.

Etapa 3: Calcula o índice do meio.

Etapa 4: Insere a chave central na árvore.

Etapa 5: Chama recursivamente:

- Subárvore esquerda: `construir_arvore(chaves, inicio, meio-1, arvore)`.
- Subárvore direita: `construir_arvore(chaves, meio+1, fim, arvore)`.

Etapa 6: Retorna a árvore construída.

Etapa 7: Finaliza liberando memória.

O código a seguir implementa essa abordagem:

```c
arvore_t *construir_arvore(int *chaves, int inicio, int fim,
    arvore_t *arvore) {
  if (inicio > fim) {
    return arvore;
  }
  int meio = (inicio + fim) / 2;
  arvore = inserir(arvore, chaves[meio]);
  arvore = construir_arvore(chaves, inicio, meio - 1, arvore
     ); // Lado esquerdo
  arvore = construir_arvore(chaves, meio + 1, fim, arvore);
        // Lado direito
```

```
  return arvore;
}
arvore_t *lista_p_arvore(int *chaves, int tamanho) {
  arvore_t *arvore = NULL;
  if (tamanho == 0) {
    return arvore;
  }
  arvore = construir_arvore(chaves, 0, tamanho - 1, arvore);
  return arvore;
}
```

3.3.3 Remoção

O processo de remoção considera três casos principais: o nó a ser removido não possui filhos, possui apenas um filho ou possui dois filhos. Quando o nó possui dois filhos, o menor elemento da subárvore direita (ou o maior da subárvore esquerda) substitui o nó removido, mantendo a organização da árvore. Para isso, utilizamos um nó auxiliar.

A seguir, apresentamos a implementação da operação de remoção:

```
arvore_t *remover(arvore_t *arvore, int chave) {
  if (arvore == NULL)
    return NULL;
  if (chave < arvore->chave) {
    arvore->esq = remover(arvore->esq, chave);
  } else if (chave > arvore->chave) {
    arvore->dir = remover(arvore->dir, chave);
  } else {
    if (arvore->esq == NULL) {
      arvore_t *temp = arvore->dir;
      free(arvore);
      return temp;
    } else if (arvore->dir == NULL) {
      arvore_t *temp = arvore->esq;
      free(arvore);
      return temp;
    }
    arvore_t *rightMin = find_min(arvore->dir);
    arvore->chave = rightMin->chave;
    arvore->dir = remover(arvore->dir, rightMin->chave);
  }
  return arvore;
}
```

3.4 Operações de Consulta

3.4.1 Busca

Na operação de busca em uma árvore binária, o valor procurado é comparado recursivamente com a chave do nó atual, começando pela raiz. Se o valor for menor

que a chave, a busca continua na subárvore à esquerda; se for maior, prossegue na subárvore à direita. Esse processo se repete até que o valor seja encontrado ou até alcançar uma folha (nó nulo), indicando que o valor não está presente na árvore.

A implementação da busca é apresentada a seguir:

```
arvore_t *buscar(arvore_t *arvore, int chave) {
  if (arvore == NULL) {
    return 0;
  }
  if (chave < arvore->chave) {
    return buscar(arvore->esq, chave);
  }
  if (chave > arvore->chave) {
    return buscar(arvore->dir, chave);
  }
  if (chave == arvore->chave) {
    return arvore;
  }
  return NULL;
}
```

3.4.2 Consulta Em Ordem

Na consulta em ordem, os nós da árvore binária são visitados seguindo a sequência: filho da esquerda, raiz e, por fim, filho da direita. Esse tipo de travessia resulta em uma listagem dos elementos em ordem crescente, caso a árvore seja uma árvore binária de busca.

```
void mostrarEmOrdem(arvore_t *arvore) {
  if (arvore != NULL) {
    mostrarEmOrdem(arvore->esq);
    printf("  %d", arvore->chave);
    mostrarEmOrdem(arvore->dir);
  }
}
```

3.4.3 Consulta Em Pré Ordem

Na consulta em pré-ordem, os nós são visitados na seguinte sequência: raiz, filho da esquerda e filho da direita. Esse método é frequentemente utilizado para gerar uma cópia da árvore ou para fins de visualização estrutural.

```
void mostrarEmOrdem(arvore_t *arvore) {
  if (arvore != NULL) {
    mostrarEmOrdem(arvore->esq);
    printf("  %d", arvore->chave);
    mostrarEmOrdem(arvore->dir);
  }
}
```

3.4.4 Consulta Em Pós Ordem

Na consulta em pós-ordem, a visita aos nós segue a ordem: filho da esquerda, filho da direita e, por último, a raiz. Esse tipo de travessia é útil em operações que envolvem a remoção ou processamento dos nós em uma sequência de base para topo (por exemplo, liberar memória ou avaliar expressões em árvores de sintaxe).

```
void mostrarPosOrdem(arvore_t *arvore) {
  if (arvore != NULL) {
    mostrarPosOrdem(arvore->esq);
    mostrarPosOrdem(arvore->dir);
    printf("  %d", arvore->chave);
  }
}
```

3.4.5 Consulta Em Nível

Para realizar uma consulta em nível, ou seja, visitar os nós da árvore da raiz até as folhas, seguindo uma ordem horizontal (nível por nível), é possível utilizar ferramentas gráficas que convertem arquivos no formato DOT para imagens, como arquivos PNG. Esse proceso permite obter a seguinte visualização da árvore:

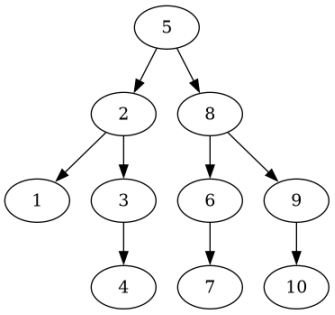

Figure 1:

4 Árvore AVL

Esta seção apresenta a implementação completa de uma Árvore AVL, esta a seguir, desenvolvida na linguagem C, incluindo suas principais operações: inserção, remoção, balanceamento, busca. Além disso, para fins de visualização, também foi implementada uma funcionalidade de exportação para formato DOT. A Árvore AVL é uma árvore binária de busca auto-balanceada que garante operações eficientes, como inserções e buscas, mantendo a complexidade $O(\log n)$.

4.1 Estrutura de Dados

A estrutura de dados da Árvore AVL (`arvore_avl`) é definida com os seguintes atributos:

```
typedef struct arvore_avl {
    int valor;
    int altura_esq;
    int altura_dir;
    struct arvore_avl *esq;
    struct arvore_avl *dir;
} arvore_avl;
```

O campo `valor` armazena o dado do nó, enquanto `altura_esq` e `altura_dir` mantêm a altura de cada subárvore. Isso facilita o balanceamento da árvore, que é realizado após cada inserção ou remoção.

4.2 Funções Principais da Árvore AVL

4.2.1 Cálculo da Altura

A função `calcular_altura` retorna a altura da subárvore. Caso o nó seja nulo (`NULL`), a altura será considerada 0. Este cálculo é essencial para determinar se a árvore precisa de balanceamento.

```
int calcular_altura(arvore_avl *arv) {
    if (arv == NULL) {
        return 0;
    }
    int altura_esq = calcular_altura(arv->esq);
    int altura_dir = calcular_altura(arv->dir);
    return (altura_esq > altura_dir ? altura_esq :
       altura_dir) + 1;
}
```

4.2.2 Criação de um Novo Nó

A função `criar_novo_no` aloca memória para um novo nó da árvore e inicializa seus campos.

```
arvore_avl *criar_novo_no(int valor) {
    arvore_avl *novo_no = (arvore_avl *)malloc(sizeof(
       arvore_avl));
    if (novo_no == NULL) {
        fprintf(stderr, "Erro ao alocar memoria para o no.\n
           ");
        exit(1);
    }
    novo_no->valor = valor;
    novo_no->altura_esq = 0;
    novo_no->altura_dir = 0;
    novo_no->esq = NULL;
    novo_no->dir = NULL;
    return novo_no;
}
```

4.2.3 Rotações de Balanceamento

O balanceamento da Árvore AVL é garantido por meio de rotações, que reorganizam os nós para manter a diferença de altura entre as subárvores esquerda e direita dentro do limite de 1.

Rotação à Esquerda A rotação à esquerda reorganiza os nós em torno do filho direito do nó desbalanceado.

```
arvore_avl *rotacao_esquerda(arvore_avl *raiz) {
    if (raiz == NULL || raiz->dir == NULL) {
        return raiz;
    }

    arvore_avl *novo_raiz = raiz->dir;
    arvore_avl *subarvore_dir = novo_raiz->esq;

    novo_raiz->esq = raiz;
    raiz->dir = subarvore_dir;

    raiz->altura_esq = calcular_altura(raiz->esq);
    raiz->altura_dir = calcular_altura(raiz->dir);

    novo_raiz->altura_esq = calcular_altura(novo_raiz->esq);
    novo_raiz->altura_dir = calcular_altura(novo_raiz->dir);

    return novo_raiz;
}
```

Rotação à Direita A rotação à direita é o análogo da rotação à esquerda, mas reorganiza os nós em torno do filho esquerdo.

```
arvore_avl *rotacao_direita(arvore_avl *raiz) {
    if (raiz == NULL || raiz->esq == NULL) {
        return raiz;
    }

    arvore_avl *novo_raiz = raiz->esq;
    arvore_avl *subarvore_esq = novo_raiz->dir;

    novo_raiz->dir = raiz;
    raiz->esq = subarvore_esq;

    raiz->altura_esq = calcular_altura(raiz->esq);
    raiz->altura_dir = calcular_altura(raiz->dir);

    novo_raiz->altura_esq = calcular_altura(novo_raiz->esq);
    novo_raiz->altura_dir = calcular_altura(novo_raiz->dir);

    return novo_raiz;
}
```

4.2.4 Inserção de um Nó

A função `inserir_no` insere um valor na árvore de maneira recursiva. Após cada inserção, a árvore é balanceada.

```
arvore_avl *inserir_no(arvore_avl *raiz, int valor) {
    arvore_avl *novo_no;

    if (raiz == NULL) {
        return criar_novo_no(valor);
    }

    if (valor < raiz->valor) {
        raiz->esq = inserir_no(raiz->esq, valor);
    } else if (valor > raiz->valor) {
        raiz->dir = inserir_no(raiz->dir, valor);
    }

    raiz->altura_esq = calcular_altura(raiz->esq);
    raiz->altura_dir = calcular_altura(raiz->dir);


    raiz = balancear_arvore(raiz);
    return raiz;
}
```

4.2.5 Remoção de um Nó

A função `remover_no` remove um valor da árvore e a rebalanceia, quando necessário.

```
arvore_avl *remover_no(arvore_avl *raiz, int valor) {
    if (raiz == NULL) {
        return NULL;
    }

    if (valor < raiz->valor) {
        raiz->esq = remover_no(raiz->esq, valor);
    } else if (valor > raiz->valor) {
        raiz->dir = remover_no(raiz->dir, valor);
    } else {
        if (raiz->esq == NULL && raiz->dir == NULL) {
        free(raiz);
        return NULL;
        } else if (raiz->esq == NULL) {
        arvore_avl *aux = raiz->dir;
        free(raiz);
        return aux;
        } else if (raiz->dir == NULL) {
        arvore_avl *aux = raiz->esq;
        free(raiz);
        return aux;
        } else {
        arvore_avl *aux = raiz->dir;
        while (aux->esq != NULL) {
            aux = aux->esq;
        }
        raiz->valor = aux->valor;
        raiz->dir = remover_no(raiz->dir, aux->valor);
        }
    }

    raiz->altura_esq = calcular_altura(raiz->esq);
    raiz->altura_dir = calcular_altura(raiz->dir);

    return balancear_arvore(raiz);
}
```

4.2.6 Busca em uma Árvore AVL

A função `buscar` localiza um valor na árvore. Caso o valor não seja encontrado, retorna `NULL`.

```
arvore_avl *buscar(arvore_avl *raiz, int valor) {

    if (raiz == NULL) {
```

```
4         return 0;
5       }
6
7       if (valor < raiz->valor) {
8         return buscar(raiz->esq, valor);
9       }
10
11      if (valor > raiz->valor) {
12        return buscar(raiz->dir, valor);
13      }
14
15      if (valor == raiz->valor) {
16        return raiz;
17      }
18      return NULL;
19  }
```

4.2.7 Exportação para DOT

A função `exportar_para_dot`, em conjunto com a função `gerar_dot`, gera um arquivo no formato DOT, possibilitando a visualização da árvore em ferramentas gráficas. Embora essa funcionalidade não seja uma característica intrínseca de uma Árvore AVL, foi incluída neste código para tornar sua análise mais acessível e intuitiva, facilitando a compreensão de sua estrutura e funcionamento.

```
1  void exportar_para_dot(arvore_avl *raiz, const char *
       nome_arquivo) {
2      FILE *arquivo = fopen(nome_arquivo, "w");
3      if (arquivo == NULL) {
4          fprintf(stderr, "Erro ao abrir o arquivo %s\n",
               nome_arquivo);
5          exit(1);
6      }
7
8      fprintf(arquivo, "digraph G {\n");
9      gerar_dot(arquivo, raiz);
10     fprintf(arquivo, "}\n");
11     fclose(arquivo);
12 }
```

4.3 Conclusão sobre a arvore AVL

A Árvore AVL é uma estrutura de dados solida e eficiente para manipulação de informações. Ela assegura que as operações de busca, inserção e remoção sejam realizadas em tempo O(logn), graças ao balanceamento automático proporcionado pelos cálculos de altura e rotações. Essa característica permite que a árvore permaneça equilibrada, independentemente da ordem em que os nós são inseridos ou removidos. Por sua eficiência, as Árvores AVL têm ampla aplicação em sistemas

que exigem alto desempenho, como bancos de dados, sistemas de arquivos e em outras diversas áreas da computação.

5 Conclusão

Nesse trabalho foi possível apanhar muito para as árvores e expandir o conhecimento sobre algoritmos de ordenação. Entender os trade-offs de cada tipo de árvore e as dificuldades para a implementação das mesmas.

O código referente ao trabalho está disponível na integra no github por meio do seguinte link:

https://github.com/RaoniSilvestre/EDB2