Segundo Trabalho Avaliativo de Estrutura de Dados Básica II

Raoni Silva, Pedro Galvão, Hélio Lima e Thiago Nascimento

January 8, 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 única unidade 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

3 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 $O(\log n)$.

3.1 Estrutura de Dados

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

```
typedef struct arvore_avl {
    int valor;                  // Valor armazenado no nó
    int altura_esq;             // Altura da subárvore esquerda
    int altura_dir;             // Altura da subárvore direita
    struct arvore_avl *esq;     // Ponteiro para o filho esquerdo
    struct arvore_avl *dir;     // Ponteiro para o filho direito
} 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.

3.2 Funções Principais da Árvore AVL

3.2.1 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;
}
```

3.2.2 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 memória para o nó.\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;
}
```

3.2.3 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;
}
```

3.2.4 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;
}
```

3.2.5 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);
}
```

3.2.6 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) {
      return 0;
    }
```

```
    if (valor < raiz->valor) {
      return buscar(raiz->esq, valor);
    }

    if (valor > raiz->valor) {
      return buscar(raiz->dir, valor);
    }

    if (valor == raiz->valor) {
      return raiz;
    }
    return NULL;
}
```

3.2.7 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.

```
void exportar_para_dot(arvore_avl *raiz, const char *nome_arquivo) {
    FILE *arquivo = fopen(nome_arquivo, "w");
    if (arquivo == NULL) {
        fprintf(stderr, "Erro ao abrir o arquivo %s\n", nome_arquivo);
        exit(1);
    }

    fprintf(arquivo, "digraph G {\n");
    gerar_dot(arquivo, raiz);
    fprintf(arquivo, "}\n");
    fclose(arquivo);
}
```

3.3 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.

4 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