Terceiro Trabalho Avaliativo de Estrutura de Dados Básica II

Raoni Silva, Pedro Galvão, Hélio Lima e Thiago Freire

January 28, 2025

Turma 35M34
Unidade 3

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.

As implementações referentes a árvore B foram desenvolvidas na linguagem Rust.

1.1 Algoritmo de Remoção

Abaixo está o fluxograma para a função de remoção de uma informação em uma árvore B.

2 Introdução

As árvores B utilizam a característica de armazenar múltiplas chaves em cada nó para organizar eficientemente os dados e os ponteiros, permitindo que operações como busca, inserção e remoção sejam realizadas de forma rápida. Além disso, sua construção garante que todas as folhas estejam sempre no mesmo nível, assegurando balanceamento e consistência na estrutura.

2.1 Propriedades

Seja d um número natural. Uma árvore B de ordem d é uma árvore ordenada que pode ser vazia ou satisfazer as seguintes condições:

- **A raiz**: É uma folha ou possui, no mínimo, dois filhos.
- **Nós internos**: Cada nó, exceto a raiz e as folhas, possui pelo menos $d + 1$ filhos.
- **Capacidade máxima**: Cada nó pode conter, no máximo, $2d + 1$ filhos.
- **Nivelamento das folhas**: Todas as folhas estão no mesmo nível, como mencionado anteriormente.

2.2 Estrutura

Para a construção da árvore B em Rust, definimos a seguinte estrutura:

2.2.1 `BTree`

A estrutura `BTree` representa a árvore principal e é definida como:

```
pub struct BTree {
    root: Node,
    grau: i32,
}
```

Os campos de `BTree` são:

- `root`: Representa a raiz da árvore B. É o ponto inicial para todas as operações, como busca, inserção e remoção.
- `grau`: Define o grau da árvore B. O grau d determina a capacidade mínima e máxima dos nós.

2.2.2 `Node`

A estrutura `Node` representa os nós da árvore B e é definida como:

```
pub struct Node {
    keys: Vec<Key>,
    children: Vec<Node>,
    is_leaf: bool,
    grade: i32,
}
```

Os campos de `Node` são:

- `keys`: Um vetor que armazena as chaves presentes no nó. As chaves são mantidas ordenadas para facilitar as operações de busca.
- `children`: Um vetor que contém os nós filhos, representando a hierarquia da árvore. Para nós folha, este vetor estará vazio.
- `is_leaf`: Um campo booleano que indica se o nó é uma folha. Nós folha não possuem filhos.
- `grade`: Representa o grau do nó, que pode ser usado para verificar o número atual de chaves ou filhos presentes.

2.2.3 `Key`

A estrutura `Key` define as informações armazenadas em cada nó e é definida como:

```
pub struct Key {
    key: i32,
    nome: String,
    quantidade: usize,
}
```

Os campos de `Key` são:

- `key`: A chave propriamente dita, utilizada para ordenação e busca na árvore.
- `nome`: Um identificador associado à chave, que pode armazenar informações descritivas ou metadados.
- `quantidade`: Representa a quantidade associada à chave.

3 Busca

O algoritmo de busca por uma chave x em uma árvore B é semelhante ao utilizado em árvores binárias de busca. No entanto, a principal diferença reside no fato de que, em uma árvore B, cada nó pode conter múltiplas chaves, tornando necessário percorrer todas as chaves presentes em um nó antes de decidir em qual subárvore continuar a busca.

A implementação em rust ficou da seguinte forma:

```
pub fn find(&self, k: Key) -> Option<Key> {
    let mut curr_node = &self.root;
    loop {
        match curr_node.search(&k) {
            SearchResult::Find(i) => return curr_node.key(i).cloned(),
            SearchResult::GoDown(i) => match curr_node.child(i) {
                None => return None,
                Some(next_node) => curr_node = next_node,
            },
        }
    }
}
```

- A função busca a chave k começando pela raiz.
- Cada nó pode ter várias chaves, então search é usado para decidir se a chave está no nó ou em qual subárvore buscar.
- Se a chave for encontrada, é retornada uma cópia dela.
- Se não for encontrada e não houver mais filhos para explorar, a função retorna None.

—

Na implementação da função search, temos:

```
pub fn search(&self, k: &Key) -> SearchResult {
    for (i, key) in self.keys.iter().enumerate() {
        match key.key.cmp(&k.key) {
            Ordering::Less => {}
            Ordering::Equal => return SearchResult::Find(i),
            Ordering::Greater => return SearchResult::GoDown(i),
        }
    }
    SearchResult::GoDown(self.keys.len())
}
```

- Retorna $Find(i)$ se a chave k for encontrada no índice i.
- Retorna $GoDown(i)$ para indicar que a busca deve continuar no filho i.

—

Esse método mantém a eficiência da busca na árvore B, garantindo um tempo de execução $O(logn)$, já que a altura da árvore é logarítmica em relação ao número de chaves armazenadas.

4 Inserção

O processo de inserção em uma Árvore B segue os seguintes passos principais:

1. **Localizar o local correto para inserir a chave:** utiliza a mesma lógica do método `search`.
2. **Inserir e balancear a árvore:** verifica se o nó é folha ou interno:
   - Se for folha, a chave é inserida diretamente.
   - Se o nó estiver cheio, ele é dividido usando a função `split`, e a chave do meio sobe para o nó pai.
   - Caso a raiz precise ser dividida, cria-se uma nova raiz, aumentando a altura da árvore.

—

4.1 Método `insert` na `BTree`

O método `insert` na `BTree` realiza a inserção na raiz da árvore, verificando se há necessidade de criar uma nova raiz.

```
pub fn insert(&mut self, k: Key) {
    let insertion = self.root.insert(k);

    if let InsertionResult::AddToFater(k, new_node) =
       insertion {
        *self = Self::new_root(self.root.clone(), k,
           new_node);
    }
}
```

- Primeiro, chama o método `insert` no nó raiz (`self.root`).
- Caso a raiz seja dividida (`AddToFater`), uma nova raiz é criada para manter a árvore balanceada.

Se a altura da árvore precisar aumentar, o método `new_root` cria uma nova raiz com dois filhos.

—

4.2 Método `insert` no `Node`

O método `insert` no `Node` realiza a inserção recursiva nos nós, seguindo as etapas abaixo:

```
pub fn insert(&mut self, k: Key) -> InsertionResult {
    let result = self.search(&k);

    if self.is_leaf {
```

```
5           return self.try_insert(k);
6       }
7
8       if let SearchResult::GoDown(i) = result {
9           let child = self.children[i].insert(k);
10
11          if let InsertionResult::AddToFater(middle_key,
                new_node) = child {
12              self.children.insert(i + 1, new_node);
13              self.children.sort();
14              self.keys.insert(i, middle_key);
15              self.keys.sort();
16
17              if self.is_full() {
18                  return self.split_node();
19              }
20          }
21      }
22
23      InsertionResult::Inserted
24  }
```

- Busca o local correto para a inserção da chave usando o método `search`.
- Se o nó for folha, insere diretamente utilizando o método `try_insert`.
- Caso contrário, chama `insert` recursivamente no nó filho correspondente.
- Após a inserção no filho:
  - Caso o nó filho seja dividido (`AddToFater`), insere o novo nó filho e a chave do meio no nó atual.
  - Se o nó atual estiver cheio, chama `split_node` para dividi-lo.

—

4.3 Função `split_node`

O método `split_node` realiza a divisão de um nó quando ele está cheio, movendo a chave do meio para o nó pai. A implementação segue:

```
1   pub fn split_node(&mut self) -> InsertionResult {
2       let t = self.grade as usize;
3       let mid = t;
4
5       let mut right_node = Node::new(self.is_leaf, self.grade)
            ;
6
7       right_node.keys = self.keys.drain(mid + 1..).collect();
8
9       if !self.is_leaf {
```

```
10          right_node.children = self.children.drain(mid + 1..)
               .collect();
11      }
12
13      let middle_key = self.keys.remove(mid);
14
15      InsertionResult::AddToFater(middle_key, right_node)
16  }
```

O método segue as etapas:

- Define `mid` como o índice central do nó.
- Cria um novo nó à direita (`right_node`).
- Move as chaves à direita do índice `mid` para `right_node`.
- Se o nó não for folha, também move os filhos correspondentes para `right_node`.
- Remove a chave do meio, que será enviada para o nó pai.
- Retorna `AddToFater`, indicando a necessidade de inserir a chave do meio no pai.

—