Universidade Presbiteriana Mackenzie

ESTRUTURA DE DADOS II

Árvore Binária de Busca (BST)
Atividade (máx. três alunos)

Objetivo

Implementar uma árvore binária de busca (BST) em Java com suporte a operações de busca, inserção e remoção e testar a sua implementação.

Instruções

- A atividade deve ser resolvida usando a linguagem Java.
- A solução não deve usar as estruturas de dados oferecidas pela linguagem Java (projetos que usarem tais estruturas serão desconsiderados – zero).
- Inclua a identificação do grupo (nome completo e RA de cada integrante) no início de cada arquivo de código, como comentário.
- Inclua todas as referências (livros, artigos, sites, vídeos, entre outros) consultadas para solucionar a atividade como comentário no arquivo `.java` que contém a `main()`.

Enunciado

1. Para a implementação da BST, você deve usar a mesma classe Java que representa um nó de uma árvore binária, criada na atividade anterior (Lab1c - Árvore Binária).

2. Crie uma classe Java que define um novo tipo de dado usado para representar uma BST (ex. `BST`). Essa classe deve ser, obrigatoriamente, uma subclasse (especialização) da árvore binária que você criou na atividade anterior (Lab1c - Árvore Binária).

A classe da BST não possui novos atributos, apenas novos métodos públicos, conforme a tabela a seguir.

OPERAÇÃO	DESCRIÇÃO
`Construtor(es)`	Construtor(es) da classe.
`search(data)`	Busca o nó com a chave `data` na BST, sendo que os nós continuam contendo apenas uma `String` na parte de dados. Caso o nó pertença à BST, retorna o nó encontrado. Caso contrário, retorna `null`.
`insert(data)`	Insere um novo nó na BST, sendo que `data` é um valor usado como chave do nó (tipo `String`). Caso já exista um nó da BST com a mesma chave, a BST não é alterada. *Opcional:* O método pode lançar uma exceção (ex. `throw new RuntimeException()`) para indicar que já existe um nó com a chave indicada.
`remove(data)`	Remove o nó com a chave `data` da BST, caso o nó pertença à BST. *Opcional:* O método pode lançar uma exceção (ex. `throw new RuntimeException()`) para indicar que a BST não possui um nó com a chave indicada.

`findMin()`	Retorna o nó com menor chave OU `null` caso a BST esteja vazia.
`findMax()`	Retorna o nó com maior chave OU `null` caso a BST esteja vazia.
`findPredecessor(data)`	Retorna o nó antecessor do nó que contém a chave indicada em `data` OU `null` caso não exista o nó com chave `data` na BST.
`findSuccessor(data)`	Retorna o nó sucessor do nó que contém a chave indicada em `data` OU `null` caso não exista o nó com chave `data` na BST.
`clear()`	Remove todos os nós da BST e suas conexões (referências para outros nós se tornam `null`).

Atenção! A tabela acima lista apenas os métodos públicos, ou seja, os métodos que a pessoa usuária da classe BST tem acesso. Caso julgue necessário, sua classe BST pode ter outros métodos auxiliares (e privados) para implementar cada operação indicada.

3. Como a chave de um nó da BST é uma string, considere a ordem alfabética para as comparações ("chave menor/maior do que o nó atual, etc.").

4. Caso julgue necessário, altere a visibilidade dos membros da classe base da árvore binária. Isto é, alterar um membro de `private` para `protected` para que a subclasse BST tenha acesso aos membros protegidos da árvore binária.

5. Para testar a sua implementação da BST, defina um conjunto de chaves (valores do tipo string) que serão inseridos na árvore. Inclua uma inserção de uma chave que já esteja presente na BST (teste de não permitir duplicatas na BST).

Realize buscas na árvore. Inclua buscas por chaves que não existem na BST. Para cada nó encontrado, exiba todas as informações do nó. Para cada nó não encontrado, exiba uma mensagem pertinente.

Remova alguns nós da árvore (pelo menos uma remoção para cada cenário estudado em aula) e tente remover nós que não existem na BST.

6. Para demonstrar o funcionamento correto da BST, para cada alteração da BST (realizada no item anterior), o programa deve exibir o conteúdo da árvore com o percurso em ordem.

"Exibir o conteúdo da árvore" significa exibir todas as informações de cada nó (chave do nó, chave do pai, chave do filho esquerdo, chave do filho direito, se é raiz, se é folha, grau, nível e altura).

Demonstre também o uso das operações `findMin()`, `findMax()`, `findPredecessor()`, `findSuccessor()` e `clear()`.

Sugestão: Você pode montar e usar a BST de exemplo apresentada no material "Árvores – Fundamentos" (conjunto de slides teóricos). Apenas lembre-se que as chaves são strings, então, no lugar de 30, você deve usar "30", no lugar de 8, deve usar "08", e assim por diante.

ESTRUTURA DE DADOS II

Entrega

Compacte o código-fonte (somente arquivos *.java) no **formato zip**.

Atenção: O arquivo zip não deve conter arquivos intermediários e/ou pastas geradas pelo compilador/IDE (ex. arquivos *.class, etc.).

Prazo de entrega: via link do Moodle até 26/03/2024 23:59.

Critérios de avaliação

A nota da atividade é calculada de acordo com os critérios da tabela a seguir.

ITEM AVALIADO	PONTUAÇÃO MÁXIMA
Implementação básica da classe que representa uma BST (subclasse da árvore binária e construtor(es)).	0,5
Implementação da operação `search(data)`.	0,5
Implementação da operação `insert(data)`.	1,0
Implementação da operação `remove(data)`.	1,5
Implementação da operação `findMin()`.	1,0
Implementação da operação `findMax()`.	1,0
Implementação da operação `findPredecessor(data)`.	1,0
Implementação da operação `findSuccessor(data)`.	1,0
Implementação da operação `clear()`.	1,0
BST de teste.	0,5
Funcionamento geral do programa, de acordo com o enunciado.	1,0

Tabela 1 - Critérios de avaliação.

A tabela a seguir contém critérios de avaliação que podem **reduzir** a nota final da atividade.

ITEM INDESEJÁVEL	REDUÇÃO DE NOTA
O projeto é cópia de outro projeto.	Projeto é zerado
O projeto usa estruturas de dados oferecida pela linguagem Java.	Projeto é zerado
Há erros de compilação e/ou o programa trava durante a execução[1].	-1,0
Não há identificação do grupo. Não há indicação de referências. Arquivos enviados em formatos incorretos. Arquivos e/ou pastas intermediárias que são criadas no processo de compilação ou pela IDE foram enviadas junto com o código-fonte.	-1,0

Tabela 2 - Critérios de avaliação (redução de nota).

O código-fonte será compilado com o compilador `javac` (21.0.2) na plataforma Windows da seguinte forma:

```
> javac *.java
```

O código compilado será executado com `java` (21.0.2) na plataforma Windows da seguinte forma (`<Classe>` deve ser substituído pelo nome da classe que contém o método `public static void main(String[] args)`):

```
> java <Classe>
```

[1] Sobre erros de compilação: considere apenas erros. Não há problema se o projeto tiver *warnings* (embora *warnings* podem avisar sobre possíveis travamentos em tempo de execução, como loop infinito, divisão por zero, etc.).

ESTRUTURA DE DADOS II

Algoritmos de busca, inserção e remoção de nós em uma BST

A seguir, são apresentados algoritmos das operações de busca, inserção e remoção de nós em uma BST. Observe que as operações podem ser implementadas de várias maneiras, sendo que os exemplos a seguir são apenas guias para começar a implementação de cada operação. Ainda, os algoritmos abaixo consideram que os nós possuem números inteiros como chave e não strings. Logo, deve ser feita uma adaptação dos algoritmos no momento da implementação da atividade, caso você use-os como referência.

Busca

```
1  // Realiza uma busca na BST.
2  Search(node: NodeBST, id: int) -> NodeBST
3    // O valor procurado não está na BST.
4    se node == null então
5      retorna null
6
7    // Encontramos o nó com o valor!
8    senão se id == node.id então
9      retorna node
10
11   // Continua busca pela esquerda.
12   senão se id < node.id então
13     retorna Search(node.left, id)
14
15   // Continua busca pela direita.
16   senão
17     retorna Search(node.right, id)
```

Inserção

```
1  // Insere um nó na BST.
2  root = Insert(root, null, value)
3
4  Insert(node: NodeBST, parent: NodeBST, id: int) -> NodeBST
5    // Achamos a posição do novo nó!
6    se node == null então
7      node = cria novo NodeBST(id, parent)
8
9    // Continua inserção pela esquerda.
10   senão se id < node.id então
11     node.left = Insert(node.left, node, id)
12
13   // Continua inserção pela direita.
14   senão se id > node.id então
15     node.right = Insert(node.right, node, id)
16
17   // Já existe um nó com o valor...
18   senão
19     não faz nada
20
21   retorna node
```

Remoção

```
// Procura o nó a ser removido da BST.
root = Remove(root, value)

Remove(node: NodeBST, id: int) -> NodeBST
  // O nó é nulo, retorna nulo.
  se node == null então
    retorna null

  // Continua busca pela esquerda.
  senão se id < node.id então
    node.left = Remove(node.left, id)

  // Continua busca pela direita.
  senão se id > node.id então
    node.right = Remove(node.right, id)

  // Encontramos o nó a ser removido!
  senão
    node = RemoveNode(node)

  // Retorna o nó, que é usado na chamada recursiva.
  retorna node
```

```
// Remove um nó da BST.
RemoveNode(node: NodeBST) -> void
  // Caso 1: o nó a ser removido é uma folha.
  se node é folha então
    node = null

  // Caso 2: o nó a ser removido não possui filho esquerdo.
  senão se node.left == null então
    node = node.right

  // Caso 3: o nó a ser removido não possui filho direito.
  senão se node.right == null então
    node = node.left

  // Caso 4: o nó a ser removido possui os dois filhos.
  senão
    // Reduzimos o problema para o Caso 3 com alguns passos extras.
    // precedessor está localizado na subárvore esquerda de
    // node e é o nó que não tem filho direito.
    predecessor: NodeBST = Predecessor(node.id)

    // Nessa versão, ao invés de atualizar as referências, estamos
    // copiando os dados do predecessor para node e removendo o nó
    // predecessor.
    node.id = predecessor.id
    node.left = Remove(node.left, predecessor.id)

  // Retorna o nó, que é usado na função Remove().
  retorna node
```