

”

**E-fólio B** | Folha de resolução para E-fólio



**UNIDADE CURRICULAR:** Estruturas de Dados e Algoritmos Fundamentais

**CÓDIGO:** 21046

**DOCENTE:** Paulo Pombinho

**A preencher pelo estudante**

**NOME:** João Paulo Alves Correia Mendes Pires

**N.º DE ESTUDANTE:** 2203810

**CURSO:** Licenciatura em Engenharia Informática

**DATA DE ENTREGA:** 07/05/2026

## RELATÓRIO:

A implementação deste programa foi realizada tendo em conta as restrições e requisitos definidos no enunciado, nomeadamente a utilização exclusiva de memória dinâmica e ponteiros normais, sem recurso a estruturas STL nem a variáveis globais, para cumprir os objetivos pretendidos.

Durante o desenvolvimento procurei aproveitar ao máximo a estrutura já existente nos ficheiros template já fornecidos, mantendo os nomes das classes, atributos e organização geral sugerida. Assim, a solução foi construída em torno da estrutura “Bnode”, responsável pela representação de cada nó da árvore, e da classe “lbst”, que representa a própria BST. A árvore possui obrigatoriamente um apontador para a raiz (*root*) e um contador do número atual de nós (*n*).

A principal referência para as opções tomadas e para a forma como foram tomadas foi o livro *Data Structures and Algorithms in C++* de Adam Drozdek, uma vez que é o recurso recomendado nesta disciplina. Em particular, os algoritmos de inserção, procura, percursos recursivos e remoção por cópia (“deletion by copying”) seguiram muito de perto a abordagem apresentada no livro, adaptando-a às exigências concretas do trabalho e aos templates fornecidos.

O programa funciona através da leitura linha a linha da entrada padrão utilizando *getline*. Cada linha é analisada através de um objeto *stringstream*, permitindo separar o comando dos seus argumentos de forma simples e eficiente. Foram ainda implementados mecanismos para ignorar corretamente linhas em branco e comentários iniciados pelo carácter #.

A operação *insert* permite inserir um ou mais valores na BST pela ordem apresentada. A inserção foi implementada de forma recursiva, descendo pela árvore até encontrar uma posição livre onde é criado um novo nó. Valores repetidos são ignorados automaticamente, preservando a propriedade da BST. A operação *find* foi implementada de forma iterativa, percorrendo a árvore através de comparações sucessivas até encontrar o elemento ou atingir uma subárvore vazia.

As operações *print\_in*, *print\_pre* e *print\_post* utilizam percursos recursivos *inorder*, *preorder* e *postorder*, respetivamente. O percurso *inorder* é importante numa BST porque produz os elementos ordenados por ordem crescente. As

operações *print\_min* e *print\_max* percorrem apenas os ramos mais à esquerda ou mais à direita da árvore, respetivamente.

A remoção de elementos foi implementada, obrigatoriamente, através do algoritmo “*deletion by copying*”, conforme especificado no enunciado. Neste algoritmo, quando o nó removido possui dois filhos, é procurado o predecessor simétrico (maior elemento da subárvore esquerda), cujo valor é copiado para o nó removido. Posteriormente, o predecessor é eliminado da sua posição original, preservando todas as propriedades da BST.

A operação *height* calcula recursivamente a altura da árvore, considerando altura -1 para uma árvore vazia, conforme exigido. Já *count\_leaves* percorre recursivamente toda a árvore contando os nós sem filhos. Relativamente à complexidade assintótica das principais operações, considerando uma árvore com altura  $h$  e  $n$  nós:

- *insert* possui complexidade  $O(h)$ , porque em cada inserção é percorrido apenas um caminho desde a raiz até uma folha;
- *find* possui complexidade  $O(h)$ , pela mesma razão;
- *delete* através de “*deletion by copying*” possui complexidade  $O(h)$ , uma vez que envolve a procura do elemento e eventualmente do predecessor simétrico;
- *print\_in* possui complexidade  $O(n)$ , porque todos os nós da árvore são visitados exatamente uma vez;
- *height* possui complexidade  $O(n)$ , dado que necessita percorrer toda a árvore para calcular a altura máxima;
- *count\_leaves* possui igualmente complexidade  $O(n)$ , pois todos os nós precisam de ser visitados para determinar quais são folhas.

No melhor caso, quando a BST se encontra equilibrada, a altura  $h$  aproxima-se de  $O(\log n)$ , tornando as operações *insert*, *find* e *delete* bastante eficientes. No pior caso, quando a árvore degenera numa lista linear, a altura passa a  $O(n)$ , degradando também essas operações para  $O(n)$ .

De uma forma geral, considero que o trabalho permitiu consolidar conceitos importantes relacionados com árvores binárias de pesquisa, recursão, memória dinâmica e manipulação de ponteiros em C++, aplicando simultaneamente algoritmos estudados ao longo da unidade curricular.