Gentian Hoxha

[Gentian.hoxha@snhu.edu](mailto:Gentian.hoxha@snhu.edu)

4/16/2024

The C++ code implements a Binary Search Tree (BST) designed for efficient management of bidding data, typical in auction systems where dynamic data handling is crucial. The BST structure is chosen for its capability to facilitate fast data retrieval, insertion, and removal, maintaining operations at logarithmic time complexity, which is essential for managing large datasets.

The BST is utilized to manage bids characterized by unique identifiers, titles, funds, and amounts, ensuring efficient and quick access to these data points. It supports operations such as inserting new bids, searching for specific bids based on identifiers, and removing bids, which are fundamental for maintaining an organized dataset in dynamic auction environments.

Developing this program involved addressing several challenges, particularly around the BST's potential imbalance, memory management in C++, and robust CSV parsing. To combat tree imbalance, foundational work was laid for potential future enhancements like integrating self-balancing AVL or Red-Black trees. Memory management was rigorously handled to avoid leaks, especially in dynamic node operations, adhering strictly to C++'s requirements for manual memory management.

Integrating CSV parsing introduced additional complexity, necessitating robust error handling to manage various input formats and data inconsistencies gracefully. This ensured the application's stability and usability, even when encountering malformed input data.

Key strategies used in the project included:

Modular Design dividing the BST into distinct methods for each operation within the `BinarySearchTree` class, the development process was streamlined, allowing for focused testing and refinement.

Memory Management Practices Implementing the Resource Acquisition Is Initialization (RAII) principle ensured efficient resource management, crucial for the program’s stability and performance.

Robust Error Handling Comprehensive error handling around CSV parsing was critical in maintaining user-friendly operation and application resilience against input errors.

Incremental Testing Each component was tested step-by-step to identify and correct errors early, ensuring reliable integration of complex operations.

In conclusion, the development of the BST for bid management was a practical exercise in applying advanced data structures in real-world applications, highlighting the importance of careful design, precise memory management, and user-centric error handling in software development.

**Tree Pseudocode**

// Define a structure for holding bid information

Structure Bid

Define bidId as String

Define title as String

Define fund as String

Define amount as Double

End Structure

// Define a node in the binary search tree

Structure Node

Define bid as Bid

Define left as Node

Define right as Node

End Structure

// Define a Binary Search Tree class

Class BinarySearchTree

Private:

Define root as Node

Public:

Constructor BinarySearchTree()

root = null

End Constructor

Destructor ~BinarySearchTree()

// Recursive delete all nodes

End Destructor

Method InOrder()

Call inOrder(root)

End Method

Method Insert(bid as Bid)

If root is null

root = new Node(bid)

Else

Call addNode(root, bid)

End If

End Method

Method Remove(bidId as String)

root = Call removeNode(root, bidId)

End Method

Method Search(bidId as String) as Bid

Define current as Node = root

While current is not null

If current.bid.bidId == bidId

Return current.bid

ElseIf bidId < current.bid.bidId

current = current.left

Else

current = current.right

End If

End While

Return new Bid

End Method

Private:

Method addNode(node as Node, bid as Bid)

If node.bid.bidId > bid.bidId

If node.left is null

node.left = new Node(bid)

Else

Call addNode(node.left, bid)

End If

Else

If node.right is null

node.right = new Node(bid)

Else

Call addNode(node.right, bid)

End If

End If

End Method

Method inOrder(node as Node)

If node is not null

Call inOrder(node.left)

Print node.bid

Call inOrder(node.right)

End If

End Method

Method removeNode(node as Node, bidId as String) as Node

If node is null

Return null

If bidId < node.bid.bidId

node.left = Call removeNode(node.left, bidId)

ElseIf bidId > node.bid.bidId

node.right = Call removeNode(node.right, bidId)

Else

If node.left is null and node.right is null

Delete node

node = null

ElseIf node.left is not null and node.right is null

Replace node with node.left

ElseIf node.left is null and node.right is not null

Replace node with node.right

Else

Find the smallest node in the right subtree

Replace node.bid with smallest bid

node.right = Call removeNode(node.right, smallest bid.bidId)

End If

End If

Return node

End Method

End Class

// Main program

Method main()

Define bst as BinarySearchTree

Define csvPath, bidKey as String

Define ticks as ClockTime

Menu operations for loading, displaying, searching, and removing bids

End Method

**References**

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/1

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/2

https://learn.zybooks.com/zybook/CS-300-R4804-OL-TRAD-UG.24EW4/chapter/5/section/8