Project Report: Binary Search Tree Implementation in Assembly Language:

Submitted by:

Name: Kaitan Das

Roll No: 21K-3002

Course: Computer Organization and Assembly Language Programming

---

1. Introduction:

This report documents the implementation of a Binary Search Tree (BST) in x86 Assembly Language using the Irvine32 library. The project demonstrates core BST operations, including insertion, searching, traversal, and deletion, while maintaining efficient memory management through dynamic heap allocation.

The goal was to create a menu-driven interactive program that allows users to perform various BST operations, showcasing an understanding of low-level programming concepts and data structure implementation in Assembly.

---

2. Project Objectives:

The primary objectives of this project were:

1. To implement a fully functional Binary Search Tree in Assembly.
2. To provide user-friendly menu navigation for BST operations.
3. To demonstrate recursive traversal techniques (Pre-order, In-order, Post-order, Level-order).

4. To handle dynamic memory allocation using Windows Heap API.
5. To ensure correctness and efficiency in BST operations.

---

3. Features Implemented:

3.1 Core BST Operations:

Operation	Description
Insertion	Adds a new node while maintaining BST properties.
Search	Checks if a value exists in the tree.
Delete Leaf Node	Removes a leaf node if found.
Find Minimum	Returns the smallest value in the tree.
Find Maximum	Returns the largest value in the tree.

3.2 Tree Traversal Methods:

Traversal	Description
Pre-order	Root → Left → Right
In-order	Left → Root → Right (ascending order)

Post-order	Left → Right → Root
Level-order (BFS)	Breadth-First Search (level by level)

3.3 User Interface:

- Interactive menu with numbered options.
- Clear prompts for user input.
- Success/error messages for operations.
- Press-any-key continuation for smooth navigation.

---

4. Technical Implementation:

4.1 Memory Management

- Heap Allocation:
  - Uses `HeapAlloc` from Windows API to dynamically allocate memory.
  - Stores nodes in a contiguous block with calculated offsets.
- Indexing:
  - Left child: `(2 * index + 1)`
  - Right child: `(2 * index + 2)`

4.2 Key Procedures

1. `Insert`
   - Recursively finds the correct position for a new node.
   - Handles duplicates by skipping insertion.
2. `Search`
   - Traverses the tree to check if a value exists.
   - Returns "Found" or "Not Found" message.
3. `Delete_Leaf`
   - Only removes nodes with no children.
   - Updates `Total_Nodes` counter.

4. Traversal Functions (`Pre_Order`, `In_Order`, `Post_Order`, `Level_Order`)
    - Implemented using recursion (except Level-order).
5. `Find_Min` & `Find_Max`
    - Traverses leftmost (min) or rightmost (max) path.

---

5. Challenges Faced:

1. Recursion in Assembly
    - Managing stack frames and registers was complex.
    - Solved by carefully preserving `ESI` (root pointer).
2. Dynamic Memory Handling
    - Calculating correct offsets for child nodes required precise arithmetic.
3. User Input Validation
    - Ensuring only valid integers were processed.
4. Debugging
    - Used OllyDbg and Visual Studio Debugger to trace register states.

---

6. Testing & Results:

- Test Cases:
    - Insertion: Verified BST property maintenance.
    - Search: Confirmed correct "Found/Not Found" responses.
    - Deletion: Ensured only leaf nodes were removed.
    - Traversals: Validated correct order outputs.

- Output Samples:

```
Breadth First Search: 50, 30, 70, 20, 40, 60, 80
In-Order Traversal: 20, 30, 40, 50, 60, 70, 80
Minimum = 20
```

- `Maximum = 80`

---

7. Learning Outcomes:

- Assembly Proficiency:
    - Improved understanding of registers, stack, and memory management.
- Algorithm Implementation:
    - Successfully implemented recursive tree traversals.
- Debugging Skills:
    - Learned to trace low-level execution flow.

---

8. Future Improvements:

1. Non-Leaf Deletion
    - Implement full node deletion (with rebalancing).
2. AVL Tree Extension
    - Add self-balancing features.
3. Graphical Visualization
    - Display tree structure in console.
4. File I/O
    - Save/load BST from a file.

---

9. Conclusion:

This project successfully demonstrates Binary Search Tree operations in Assembly Language, proving that complex data structures can be efficiently implemented at a low level. The menu-driven interface ensures usability, while the dynamic memory management highlights optimization skills.

This implementation serves as a strong foundation for further exploration in low-level data structures and performance-critical applications.

---

10. References:

- Irvine32 Library Documentation
- Intel x86 Assembly Manual
- Windows Heap API Guide

Submitted by:

Kaitan Das

Roll No: 21K-3002

Date: 30-07-2025