**Dynamic Pointer-Based Binary Search Trees**

Mohammed Rayaan Pasha | *Bengaluru, India* | [mdrayaanpasha@gmail.com](mailto:mdrayaanpasha@gmail.com)

| ***Abstract* – *Efficient management of data structures poses significant challenges in terms of time and space complexity, particularly for operations such as searching, inserting, and deleting. While Binary Search Trees (BSTs) and hash tables offer potential solutions, they each come with limitations: BSTs can exhibit O(log n) complexity, and hash tables face issues such as collisions. This paper presents a new data structure that combines pointer machines with binary search trees to address these challenges. The proposed structure uses the strengths of both components to enhance efficiency in data management. By combining the dynamic capabilities of pointer machines with the balanced nature of BSTs, the new data structure achieves improved performance in terms of both time and space complexity.***   1. **INTRODUCTION**   Managing data in a data structure is a classic problem; the effectiveness of a data structure is commonly judged by its searching, inserting, and deleting time and space complexity. Binary search trees and hash tables are attractive solutions for this problem, but these solutions have some disadvantages.  While BST's ensure O (log n) average time complexity, hash tables can be more efficient with constant time complexity in the average case. But they have problems with collisions, and to improve them, they just increase the space complexity.  To find a solution for this, this paper introduces a new data structure, Pointer-Enhanced BST's, that combines binary search trees with pointer machines in a clever way to gain efficiency with an average time complexity of O(log(Query Number) + log k), where k is the number of keys in the node containing the key in the large data set. e.g., if the data to search is 16, then it will be: O(log 16 + log 8) 4 + 3 = 7 in the worst case; that is better than the worst case of BST's.  The clever way is: in our data structure, we use nodes with increasing capacities for storing elements. Initially, each node has a capacity to store exactly 2 elements. For instance, a node might store the elements {1, 2}. As the need for additional storage arises, the capacity of subsequent nodes is doubled. Thus, the next node in the sequence can accommodate up to 4 elements and may contain { 3, 4}. This pattern continues, with each successive node doubling its capacity compared to its predecessor. Therefore, a node with a capacity of 8 might store {5, 6, 7, 8}. This process of doubling the capacity and storing elements proceeds indefinitely, allowing the structure to accommodate an ever growing sequence of elements.    **III. Methodology:**  We merge the concepts of BST’s with Pointer Machines. Basic idea is to add binary search trees in the nodes of the pointer machine.  Suppose a node in pointer machine has k nodes, then the next node must have k\*2 nodes. And a node in pointer machine has the stores that has keys k/2…k/2 + 1. ….. k-1.  For eg: suppose k=4 for first node:  Then it can store values from k/2 k/2 + 1….. k-1 nodes inside the BST of the node. Meaning k/2 here is 2 and k is 4 so the bst in this node has 2,3.  Now lets suppose this for a large number like k=128. Then the pointer machine node with k=128 stores value in its bst from k/2 that is 64 to k-1 that is 127. It would be like: 64,65,67…….127   1. **Design of Data Structure**   Node Structure:   * Linked List L: * Each node N has a capacity k where k doubles for each subsequent node. * Each node N contains a BST B with a capacity of k/2, storing keys in the range [k/2, k-1].  1. **Search Algorithm**  * Input: Key x * Initialize: Set A = 1. * Traverse: Iterate through the linked list L. * Condition: If k/2 <= x <= k-1 at node N with capacity k: * Search: Perform BST search within B at node N. * if Found: Return success. * Else: Set A = A \* 2 and continue. * If End of List Reached: Return "not found."  1. **Insertion Algorithm**      * Input: Key x * Initialize: Set A = 1. * Traverse: Iterate through the linked list L. * Condition: If k/2 <= x <= k-1 at node N with capacity k: * Insert: Insert x into BST B at node N. * If Insertion Successful: Return success. * Else: Set A = A \* 2 and continue. * If End of List Reached: * Create Node: Generate new nodes in L with capacities doubling from previous k until a suitable position is found. * Insert: Insert x into the BST of the newly created node. * Return: Success.  * O(floor(log(n+1)))+O(floor(log(n+1)))=O(2∗floor(log(n+1)))O(floor(log(n + 1))) + O(floor(log(n + 1))) = O(2 \* floor(log(n + inserting keys (insert) and searching for keys (search). The insert method ensures that keys are added to the correct position in the tree, while the search method traverses the tree to locate a key.1)))O(floor(log(n+1)))+O(floor(log(n+1)))=O(2∗floor(log(n+1)))   Thus, by the principle of mathematical induction, the claim holds for all queryKey ≥ 2. **2. Proof by Contradiction****Assumption:** Suppose, for the sake of contradiction, that the time complexity for searching, inserting, and deleting in the DPB-BST is less than O(2 \* floor(log(queryKey))), i.e., O(floor(log(queryKey))). **Analysis:**  * **Traversal of Linked List:**   + The structure of DPB-BST involves traversing through nodes whose capacities double at each step. This traversal process naturally requires O(floor(log(queryKey))) steps to reach the correct node.   + Reducing this to a lower complexity implies that fewer nodes are traversed, contradicting the node doubling pattern that defines the DPB-BST. * **BST Search:**   + Once the correct node is reached, searching within the BST of that node inherently requires O(floor(log(queryKey))), as it follows the logarithmic nature of binary search.   + Reducing this step further would imply a violation of the fundamental properties of binary search, which cannot be achieved without altering the basic structure.  **Contradiction:** The assumption that the time complexity is less than O(2 \* floor(log(queryKey))) contradicts the necessity of performing both traversal and BST search, each requiring O(floor(log(queryKey))). **Conclusion:** Thus, by contradiction, the time complexity for searching, inserting, and deleting in the DPB-BST must be O(2 \* floor(log(queryKey))).  **V. Implementation & Result**  In this section, we describe the implementation of the Dynamic Pointer-Based Binary Search Tree. The implementation is divided into several key components:  **1. Classes and Components:**  The results indicated that all 200 inserted keys were correctly found, while non-existing keys were appropriately identified as absent. The recorded execution time aligns with the theoretically expected performance, supporting the correctness of the data structure’s operations. Profiling results confirmed that the function calls performed efficiently, consistent with the theoretical time complexity.  The time complexity of O(2 × log(queryKey) ) was mathematically proven prior to experimentation. The experimental results validate this theoretical performance, demonstrating that the data structure performs effectively for the tested scale and adheres to the predicted complexity.  **VI. Conclusion:**  The Dynamic Pointer-Based Binary Search Tree (DPB-BST) demonstrates a significant advancement in data management by integrating Binary Search Trees (BSTs) with pointer machines. This innovative data structure enhances both time and space efficiency by leveraging the strengths of its components. The key advantage of the DPB-BST is its improved time complexity, which has been theoretically proven to be O(2 × log(queryKey)⌋). This performance improvement is achieved through a novel design where each node in the pointer machine contains a BST with dynamically doubling capacities, ensuring that operations remain efficient even as the dataset grows.  The experimental results validate the theoretical performance of the DPB-BST. A series of tests involving the insertion and search of 200 unique keys demonstrated that the structure handled operations accurately and efficiently. All existing keys were successfully located, and non-existing keys were correctly identified as absent. The execution time recorded during these experiments was consistent with the predicted complexity, confirming that the DPB-BST performs as expected in practical scenarios. | Visiuall Representation of Dynamic Pointer-Based Binary Search Trees   1. **Background Study**   **Binary Search Trees:** they are a type of data structure where each node is connected to two children, and the value of the left child is less than the parent node, while the right child has a greater value. This setup helps in quickly finding, adding, or removing values, usually taking O(log n) time for these operations. but, when the BST becomes unbalanced, it can take O(n) time, which is less efficient. To solve this, there are balanced BSTs like AVL trees that keep the tree balanced and be efficient.  **Hash Tables:**  work differently by using a hash function to map data to specific locations in an array for very fast operations, with an average time complexity of O(1) for searching, adding, and removing data. However, hash tables face problems like collisions and resizing. techniques like chaining, open addressing & table doubling help manage collisions, but they don't completely eliminate these issues.  **Pointer Machines** are a theoretical concept used to manage data structures more flexibly, instead of using fixed memory locations, pointer machines use pointers to handle data, making them good for dynamic data arrangements. They help in frequent adjustments for varying sizes.  **Combining BSTs and Pointer Machines** is an idea that hasn’t been fully explored yet. The integration could potentially combine the strengths of both: the organized structure of BSTs and the dynamic handling of pointer machines. This combination could address some of the problems faced by BSTs and hash tables, like inefficient performance when trees are unbalanced or issues with collisions.  **D. Deletion Algorithm**   * Input Key x * Initialize: Set A = 1 * Traverse: Iterate through the linked list L. * Condition: If k/2 <= x <= k-1 at node N with capacity k: * Search: Search for x in BST B at node N. * If Found: Delete x from BST B. * If BST Becomes Empty: Consider removing node N. * Return: Success. * Else: Set A = A \* 2 and continue. * If End of List Reached: Return "not found."   **IV. Mathematical Proof For Time Complexity.** **1. Proof by Induction****Claim:** The time complexity for searching, inserting, and deleting in the Dynamic Pointer-Based Binary Search Tree (DPB-BST) is O(2 \* floor(log(queryKey))). **Base Case:** For queryKey = 2:   * The initial node in the DPB-BST has a size of at least 2. * Searching, inserting, or deleting within this node involves examining the node itself, which takes O(1), and performing a binary search within the BST of size 2, also taking O(1). * Therefore, the total time complexity is O(2 \* floor(log 2)) = O(2 \* 1) = O(2).   Thus, the claim holds for the base case. **Inductive Hypothesis:** Assume the time complexity is O(2 \* floor(log(queryKey))) for all queryKey ≤ n, where n ≥ 2. **Inductive Step:** Consider the case where queryKey = n + 1:   * **Traversal of Linked List:**   + To locate the node containing queryKey = n + 1, the traversal progresses through nodes with capacities 2, 4, 8, …, up to the node with the appropriate capacity for n + 1.   + The number of such steps corresponds to O(floor(log(n + 1))). * **BST Search:**   + Once the correct node is found, a search within the BST of that node takes O(floor(log(n + 1))). * **Total Time Complexity:**   - Node Class  - Represents an individual node in the binary search tree (BST). It contains attributes for the node’s data, left child, and  due to the traversal of the linked list and the logarithmic search within the BST.  - BST Class:  - Implements the BST operations.    - PMachine Class:  - Represents a node in the pointer machine, which contains a BST and a link to the next PMachine node. The max\_num attribute defines the maximum value that this node can handle.  - PointerMachineHandler Class:  - Manages the linked list of PMachine nodes. It includes methods for inserting keys (insert) and searching for keys (search). The insert method traverses the linked list, creates new PMachine nodes if necessary, and inserts keys into the appropriate BST. The search method traverses the linked list to locate the correct PMachine and performs a search within the associated BST.  **2. Implementation Details:**  - Initialization  - The PointerMachineHandler class initializes with an initial limit for the first PMachine node. The linked list of PMachine nodes starts with this initial limit and can grow as needed.  - Insertion Process  - To insert a key, the handler traverses the linked list of PMachine nodes to find the correct node based on the key. If necessary, new PMachine nodes are created with doubled capacities. The key is then inserted into the BST of the appropriate PMachine.  - Search Process  - To search for a key, the handler traverses the linked list to locate the correct PMachine    **3. Edge Case Handling**  - The implementation handles cases where the key requires creating new PMachine nodes by doubling the capacity as needed. If a key is not found in any PMachine, a  **5. Experimental Results**    The implementation was evaluated to assess its performance for key insertion and search operations. A total of 200 unique keys were randomly generated and inserted into the structure, and both existing and non-existing keys were subsequently searched. The execution time for these operations was measured, and detailed performance metrics were gathered using Python’s cprofile module. |
| --- | --- |
|  |  |