Analysis and Comparision of Data Structures

Muneeba Badar (08091)
Syeda Samah Daniyal (07838)
Fakeha Faisal (08288)
Eman Fatima (08595)
Qurba Mushtaque (08232)

Dhanani School of Science and Engineering, Habib University
CS 201 - Data Structures II
L5
Mr. Abdullah Zafar
April 29, 2024

Introduction

Our project delves into the comparative analysis of three tree data structures—**Binary Search Trees (BSTs), Adelson-Velsky and Landis Trees (AVL), and Treaps**—focusing on their search and insert functions. We meticulously crafted both standard and optimized versions of these functions and evaluated their performance using diverse datasets. Through systematic experimentation and graphical analysis, we aimed to discern the efficacy of optimization techniques in enhancing runtime efficiency. By documenting our optimization strategies and analyzing the results, we sought to provide valuable insights into the impact of optimization on tree data structure performance.

Binary Search Tree (BST)

2.1 Search Function

2.2.1 Original code

The original search function that we implemented was using recursion to search a particular word within the BST. If the word's first alphabet comes before the current word's first alphabet, indicating it lies in the left subtree, the function recursively calls itself with the left subtree. Conversely, if the word's first alphabet comes after the current word's first alphabet, implying it lies in the right subtree, the function recursively calls itself with the right subtree. This recursive approach enables traversal through the tree until the desired word is found or the end of the tree is reached.

```
if (Key == node->key) {
    return node->indices;
} else if (Key < node->key) {
    return searchHelper(node->left, Key);
} else {
    return searchHelper(node->right, Key);
}
```

Fig 2.1.1: Recursive Search

2.1.2 Optimized Version

In our initial attempt to optimize the code, we opted for a while loop over recursion, drawing from prior research indicating that loops tend to outperform recursion on larger datasets. This choice led us to employ a while loop for traversing the Binary Search Tree (BST) in search of the target word. Within the loop, the current node is systematically updated to either its left or right child, depending on the comparison with the target word. This iterative approach facilitated efficient traversal through the BST, aligning with our aim to enhance performance on sizable datasets.

```
while (node != nullptr) {
    if (Key == node->key) {
        return node->indices;
    }
    else if (Key < node->key) {
        node = node->left;
    }
    else {
        node = node->right;
    }
}
```

Fig 2.1.2a: While loop for traversal

In our second iteration of optimizing the code, we retained the while loop but strategically positioned the check for if (node == nullptr) outside the loop at the top. This avoids unnecessary comparisons and loop iterations if the starting node is already nullptr. Furthermore, we enhanced the comparison logic by employing a conditional ternary operator, streamlining the code and bolstering its efficiency. These refinements collectively aimed to maximize the performance of the code, particularly in scenarios involving large datasets.

```
if (node == nullptr) {
  return vector<int>(); // Early return if node is null
}

// Optimized comparison: avoid string copy in each iteration
while (node != nullptr && Key != node->key) {
  node = (Key < node->key) ? node->left : node->right;
}
```

Fig 2.1.2b: Ternary operator logic

2.1.3 Changes and Impact

Switching the function from a recursive approach to a loop approach decreased the time in nanoseconds to search the words by their name in the given BST. The iterative method demonstrated superior speed, attributed to its fast execution without the overhead of function calls typical in recursion. Its time complexity also decreased compared to the recursive counterpart, ensuring more efficient traversal and search operations. Moreover, by removing the need of additional data structures like stacks, the iterative approach conserved memory space and

proved particularly advantageous for handling larger datasets. These changes also made it less prone to crashes caused by recursive function calls.

Before:

```
return searchHelper(node->left, Key);
```

```
return searchHelper(node->right, Key);
```

After:

```
node = node->left;
```

```
node = node->right;
```

Opting for a ternary operator rather for the comparison also eliminated the need to create a temporary string object for comparison in each iteration, not only reducing the memory overhead and potential string copying but also enhancing performance by lowering memory usage.

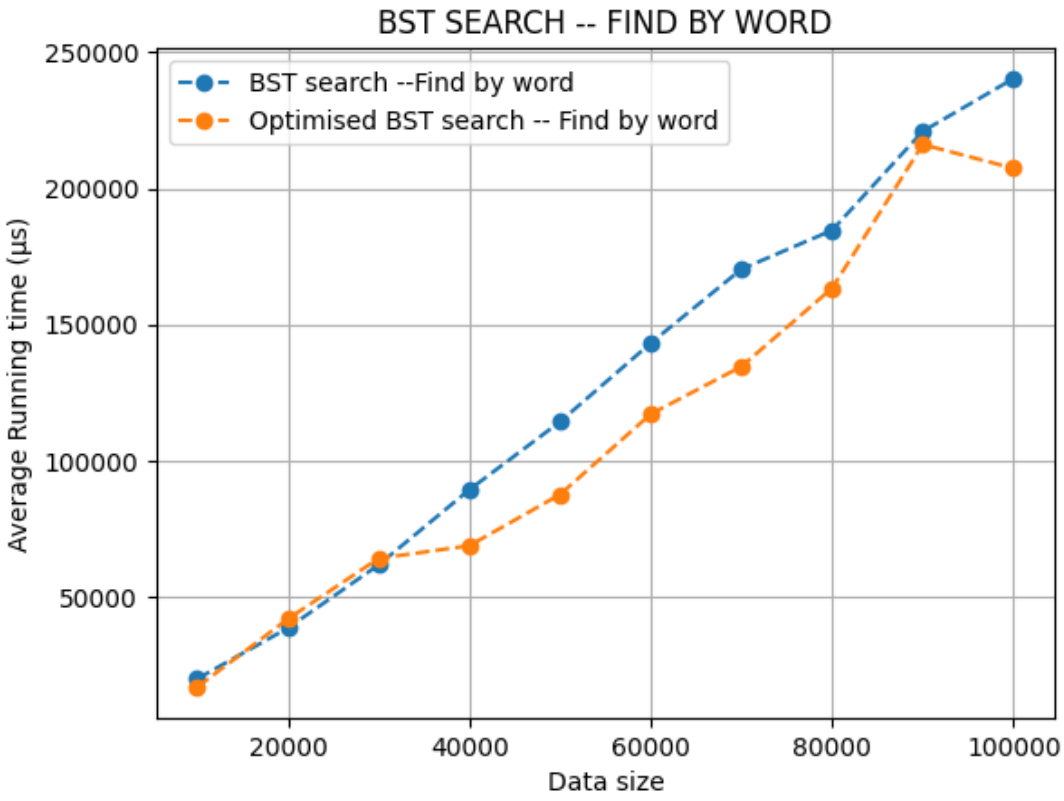
Before:

```
else if (Key < node->key) {
    node = node->left;
}
else {
    node = node->right;
}
```

After:

```
node = (Key < node->key) ? node->left : node->right;
```

2.1.4 Testing and Validation



2.2 Insert Function

2.2.1 Original code

The original insertion function, much like the search function, employed recursion to insert nodes into the left or right subtree. It iteratively delved into the left or right subtree by comparing the first letter of the current word with the first letter of the word to be inserted.

```
else if (key < node->key) {
    // Recursively insert into the left subtree
    node->left = insertHelper(node->left, key, i);
}
else if (key > node->key) {
    // Recursively insert into the right subtree
    node->right = insertHelper(node->right, key, i);
}
```

Fig 2.2.1: Recursive Insertion

2.2.2 Optimized Version

In the optimized version, we've replaced recursive calls with a while loop for iterative tree traversal during node insertion. This reduces function call overhead. Additionally, we've added break statements after successful insertions to avoid unnecessary iterations through the loop. These improvements streamline the insertion process for better efficiency.

```
while (current != nullptr) {
  if (key < current->key) {
    if (current->left == nullptr) {
      current->left = new Node(key, i);
      break; // Insertion complete
    }
    else {
      current = current->left;
    }
```

Fig 2.1.2a: while loop

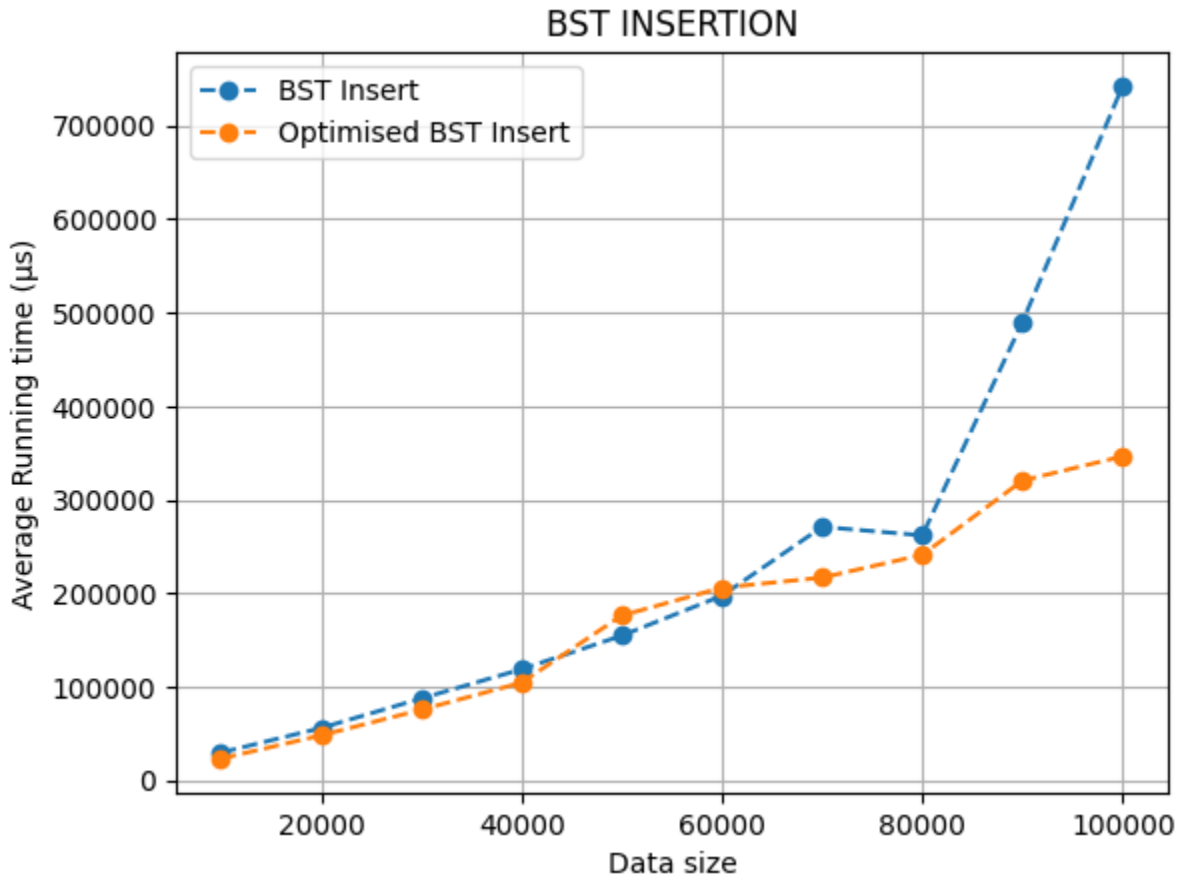
```
else if (key > current->key) {
  if (current->right == nullptr) {
    current->right = new Node(key, i);
    break; // Insertion complete
  }
  else {
    current = current->right;
  }
}
```

Fig 2.1.2b: Early break statement

2.2.3 Changes and Impact

The recent modifications have significantly enhanced the insertion process. The addition of break statements after successful insertions streamlines traversal, reducing unnecessary iterations. This cuts down on time complexity. Additionally, transitioning from recursive calls to iterative traversal decreases memory overhead, particularly for deep trees and large datasets. This minimizes the risk of stack overflow errors, ensuring greater stability in handling extensive data structures.

2.2.4 Testing and Validation



Adelson-Verlsky and Landis Tree (AVL)

3.1 Search function

3.1.1 Original Code

In the original code, when searching for a key, we first verify if a node exists for that key. If no node is found, we create an empty vector and return it.

```
vector<int> AVLIndex::searchHelper(Node* node, const string& Key) {
        if (node == nullptr) {
            return vector<int>();
        }
```

Fig 3.1.1 Returning an empty vector

3.1.2 Optimized Code

In the optimized version, if no node exists, I simply return an empty vector {} instead of creating a new one.

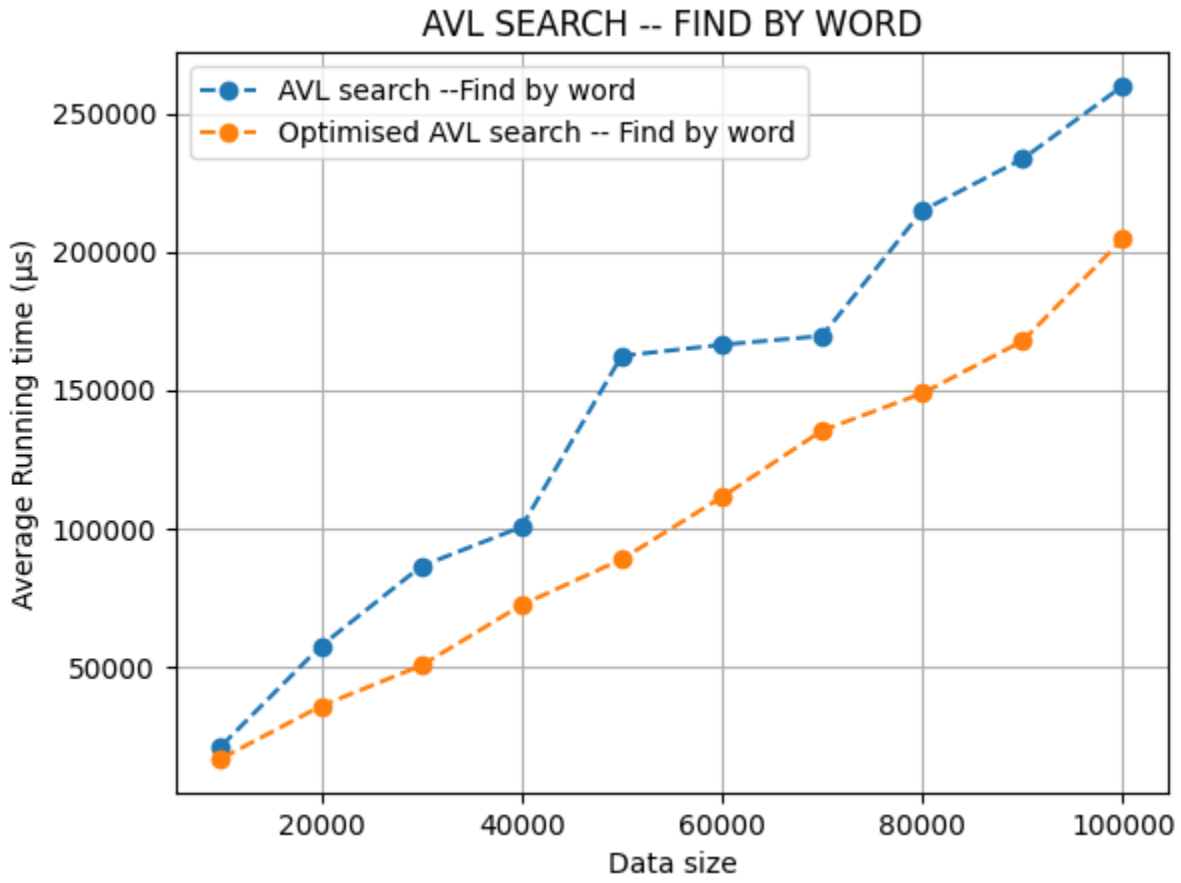
```
vector<int> AVLIndex::searchHelper(Node* node, const string& Key) {
    if (node == nullptr) {
        return vector<int>();
    }
```

Fig 3.1.2 Returning curly braces {}

3.1.3 Changes and Impact

In the first code snippet, return vector<int>();, a new empty vector is explicitly created and returned. However, in the second code snippet, return {};, no new vector is explicitly created. Instead, an empty vector is constructed using the list initialization syntax {} and returned directly. From a performance perspective, both approaches should have negligible differences. The second approach may save a minuscule amount of processing time by avoiding the explicit creation of a new vector, but the difference is likely insignificant in most scenarios.

3.1.4 Testing and Validation



3.2 Insert function

3.2.1 Original Code

Balance Factor

In the original code, we're storing the heights of the left and right subtrees in separate variables, and then using it to calculate the balance factor.

```
int AVLIndex::balanceFactor(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        return getHeight(node->left) - getHeight(node->right);
    }
```

fig 3.2.1 Balance factor

Right rotation:

In this snippet, we execute a basic right rotation by promoting the left child to the parent position, effectively demoting the parent to the right child of the former left child, which now assumes the role of the parent node.

```
Node* AVLIndex::rightRotate(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;

        // Perform rotation
        x->right = y;
        y->left = T2;

        // Update heights
        y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
        x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

        return x;
    }
```

Fig 3.2.1 b Right Rotation

Left rotation:

In this snippet, we execute a basic Left rotation by promoting the right child to the parent position, effectively demoting the parent to the left child of the former right child, which now assumes the role of the parent node.

```
Node* AVLIndex::leftRotate(Node* x) {
        Node* y = x->right;
        Node* T2 = y->left;

        // Perform rotation
        y->left = x;
        x->right = T2;

        // Update heights
        x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
        y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

        return y;
}
```

Fig 3.2.1 c Left Rotation

Insert helper function:

In this function, we start by comparing the given key with the key of the parent node to determine the direction in which the new node should be inserted. Once the new node is inserted, we update its height and calculate its balance factor. Based on the balance factor and the value of the key being inserted, we perform rotations to ensure that the AVL tree property is maintained.

```
Node* AVLIndex::insertHelper(Node*& node, const int i, const string& Key) {
        // Perform normal BST insertion
        if (node == nullptr) {
            return new Node(Key, i);
        }
        if (Key < node->key) {
            node->left = insertHelper(node->left, i, Key);
        } else if (Key > node->key) {
            node->right = insertHelper(node->right, i, Key);
        } else {
            // Key already exists, add index to the existing node
            node->indices.push_back(i);
        }
        // Update height of this ancestor node
        node->height = 1 + max(getHeight(node->left), getHeight(node->right));
        // Get the balance factor of this ancestor node to check whether this node became unbalanced
        int balance = balanceFactor(node);
        // If this node becomes unbalanced, then there are four cases
        // Left Case
        if (balance > 1 && Key < node->left->key) {
            return rightRotate(node);
        }
        // Right Case
        if (balance < -1 && Key > node->right->key) {
            return leftRotate(node);
        }
        // Left Right Case
        if (balance > 1 && Key > node->left->key) {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
        // Right Left Case
        if (balance < -1 && Key < node->right->key) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
        return node; // Return the (unchanged) node pointer
}
```

Fig 3.2.1 d: insert helper function

3.2.2 Optimized Code

Balance Factor:

In this version, I've simplified the process by directly computing the balance factor using the height of the left and right subtrees. This eliminates the need for separate variables to store the heights of the left and right subtrees.

```
int AVLIndex:: getBalanceFactor(Node* node) {
    return (node == nullptr) ? 0 : getHeight(node->left) - getHeight(node->right);
}
```

Fig 3.2.2 a balance factor

Right rotation:

Rather than immediately proceeding with rotation, I first check whether the parent node or its child is empty. If either of them is empty, I simply return the parent node without performing any rotation. This approach helps conserve the time that would otherwise be spent on the entire rotation process.

```
Node* AVLIndex::rightRotate(Node* y) {
    if (y == nullptr || y->left == nullptr) {
        return y; // Return y unchanged if it's null or its left child is null
    }

    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    // Update heights after rotation
    y->height = 1 + max(getHeight(y->left), getHeight(y->right));
    x->height = 1 + max(getHeight(x->left), getHeight(x->right));

    return x; // Return the new root of the subtree
}
```

Fig 3.2.2 b right rotation

Left rotation:

Here, instead of introducing a pointer `T` to reference `y->left`, then assigning `x->right` to `T2`, I've directly set `x->right` to `y->left`. This optimization eliminates the need for an extra pointer variable.

```
Node* AVLIndex::leftRotate(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;

    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}
```

Fig 3.2.2 c left rotation

Insert helper function:

In original code we had two condition that were checking if the balance factor is –1 and two conditions that checked if the balance factor is 1 and then performed rotation accordingly but i changed code by making nested condition first checking if the balance factor is 1 if yes then we check 2 condition and if it's not equal to 1 then it skips both the condition and directly move to another condition block in this way we are not wasting our time checking each condition.

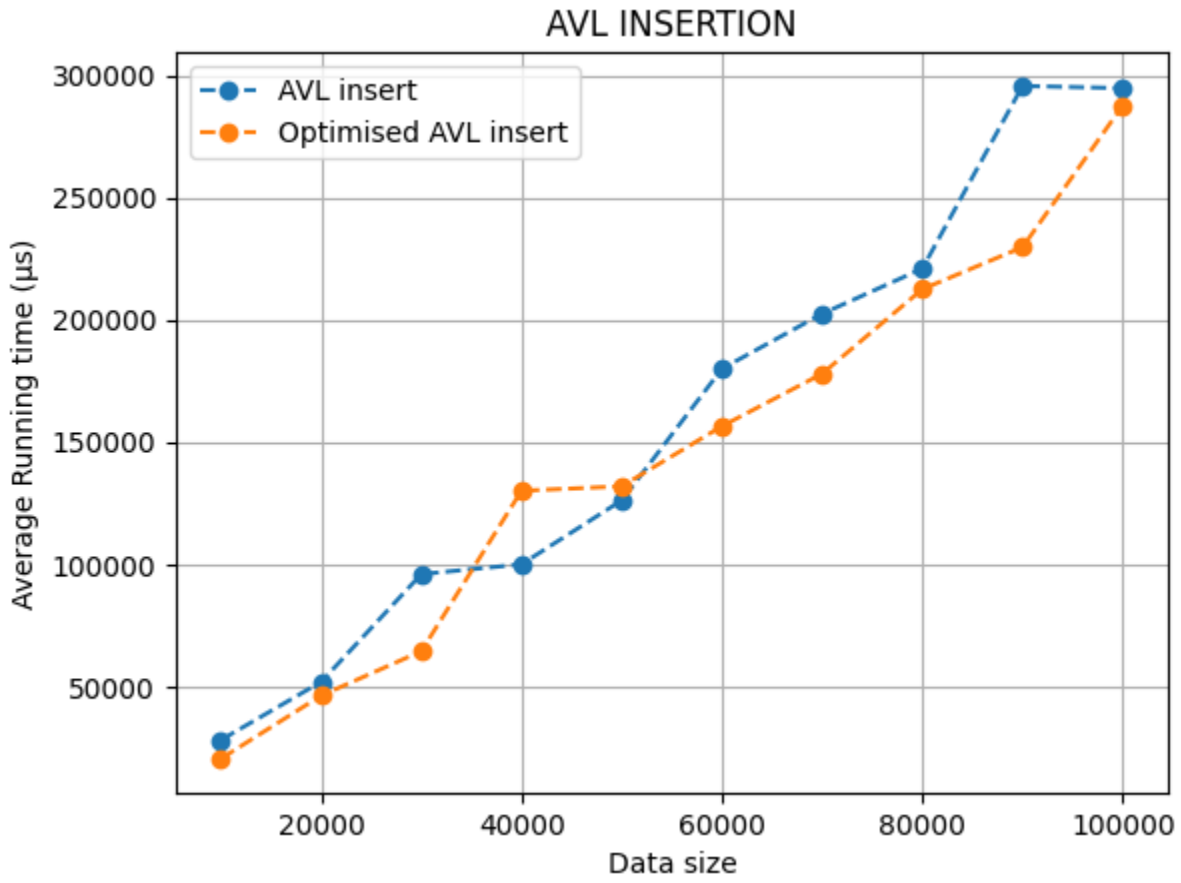
```
Node* AVLIndex::insertHelper(Node*& node, const string& Key, const int i) {
    if (node == nullptr) {
        return new Node(Key, i);
    }
    if (Key < node->key) {
        node->left = insertHelper(node->left, Key, i);
    } else if (Key > node->key) {
        node->right = insertHelper(node->right, Key, i);
    } else {
        node->indices.push_back(i);
        return node;
    }
    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int balance = getBalanceFactor(node);
    if (balance > 1) {
        if (Key < node->left->key) {
            return rightRotate(node);
        } else {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
    }
    if (balance < -1) {
        if (Key > node->right->key) {
            return leftRotate(node);
        } else {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
    }
    return node;
}
```

Fig 3.2.2 d insert helper

3.2.3 Changes and Impact

The new optimized code outperforms the original version significantly by eliminating extra variables, pointers, and unnecessary condition checks. This optimization strategy has led to substantial time savings in the program's execution.

3.2.4 Testing and Validation



Treaps

4.1 Search function

4.1.1 Original code version

The original **searchHelper** function employs a recursive approach to traverse the Treap, checking for null nodes and comparing keys for the target. Duplicate key handling and tree traversal are integrated into the recursive calls, making the code less modular and potentially harder to understand.

```
} else if (Key < node->key) {
    return searchHelper(node->left, Key);
} else {
    return searchHelper(node->right, Key);
}
```

Fig 4.1.1: Recursive Call Treaps

4.1.2 Optimized code version

The optimized **searchHelper** function improves time complexity by replacing recursion with an iterative approach

```
while (node != nullptr) {
    if (Key == node->key) {
        return node->indices;
    } else if (Key < node->key) {
        node = node->left;
    } else {
        node = node->right;
    }
}
return vector<int>(); // Key not found
```

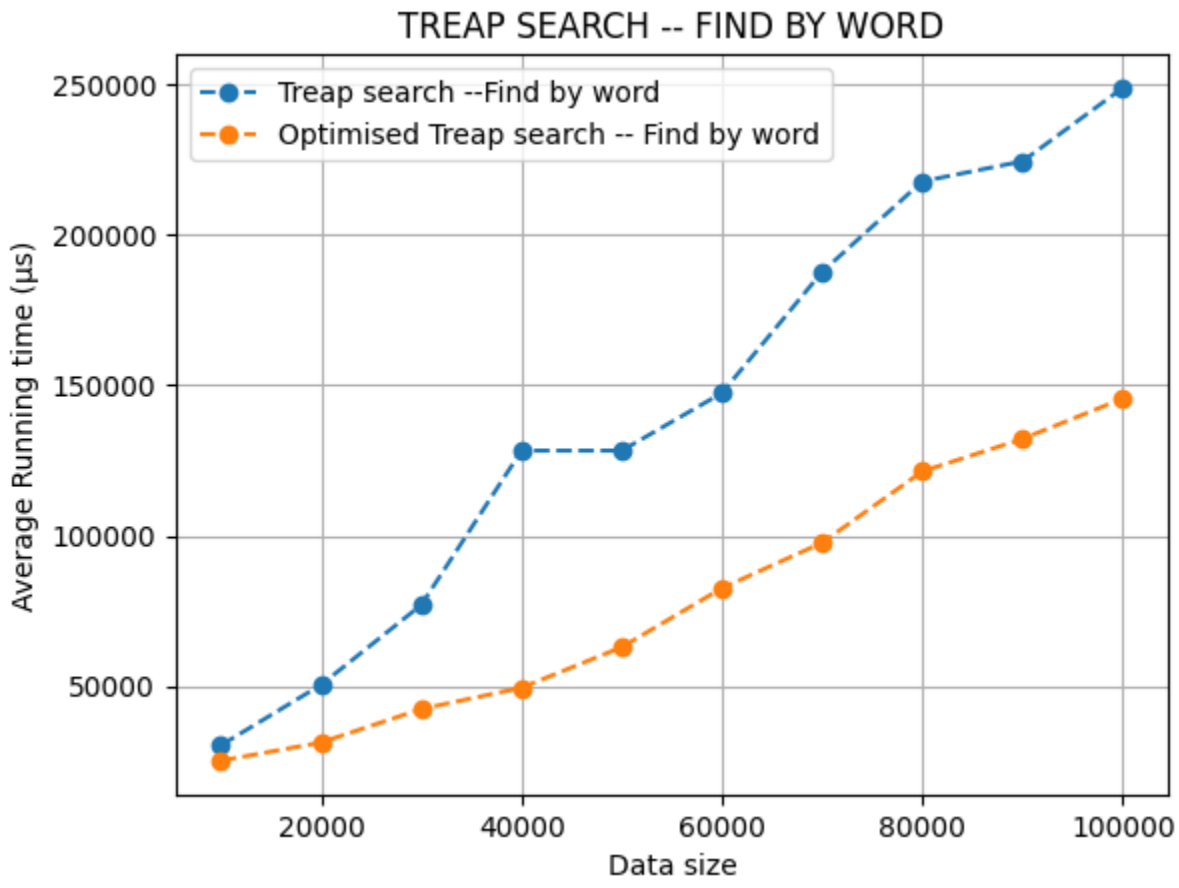
Fig 4.1.2: While Loop for Treap

4.1.3 Changes and Impact

Replacing recursion with an iterative approach reduced the risk of stack overflow and improved overall efficiency. With separate key comparison and traversal, the optimized version offers a

more streamlined search process, potentially leading to faster search times, particularly for large datasets. Additionally, the removal of recursion may reduce memory overhead, contributing to improved performance and scalability of the Treap data structure.

4.2.4 Testing and Validation



4.2 Insert function

4.2.1 Original code

The original `insertHelper` function utilizes recursion to insert nodes into the Treap, which may lead to stack overflow issues with large datasets. Duplicate key handling and tree rotations are integrated into the recursive calls, making the code less modular and potentially harder to understand. This recursive approach might hinder scalability and efficiency, particularly when dealing with significant amounts of data.

```
else if (key < node->key) {
    // Recursively insert into the left subtree
    node->left = insertHelper(node->left, key, i);
    if (node->left->p > node->p)
        node = rightRotate(node);
}
else if (key > node->key) {
    // Recursively insert into the right subtree
    node->right = insertHelper(node->right, key, i);
    // Fix Heap property if it is violated
    if (node->right->p > node->p)
        node = leftRotate(node);
}
```

Fig 4.2.1: Recursive calls

4.2.2 Optimized code

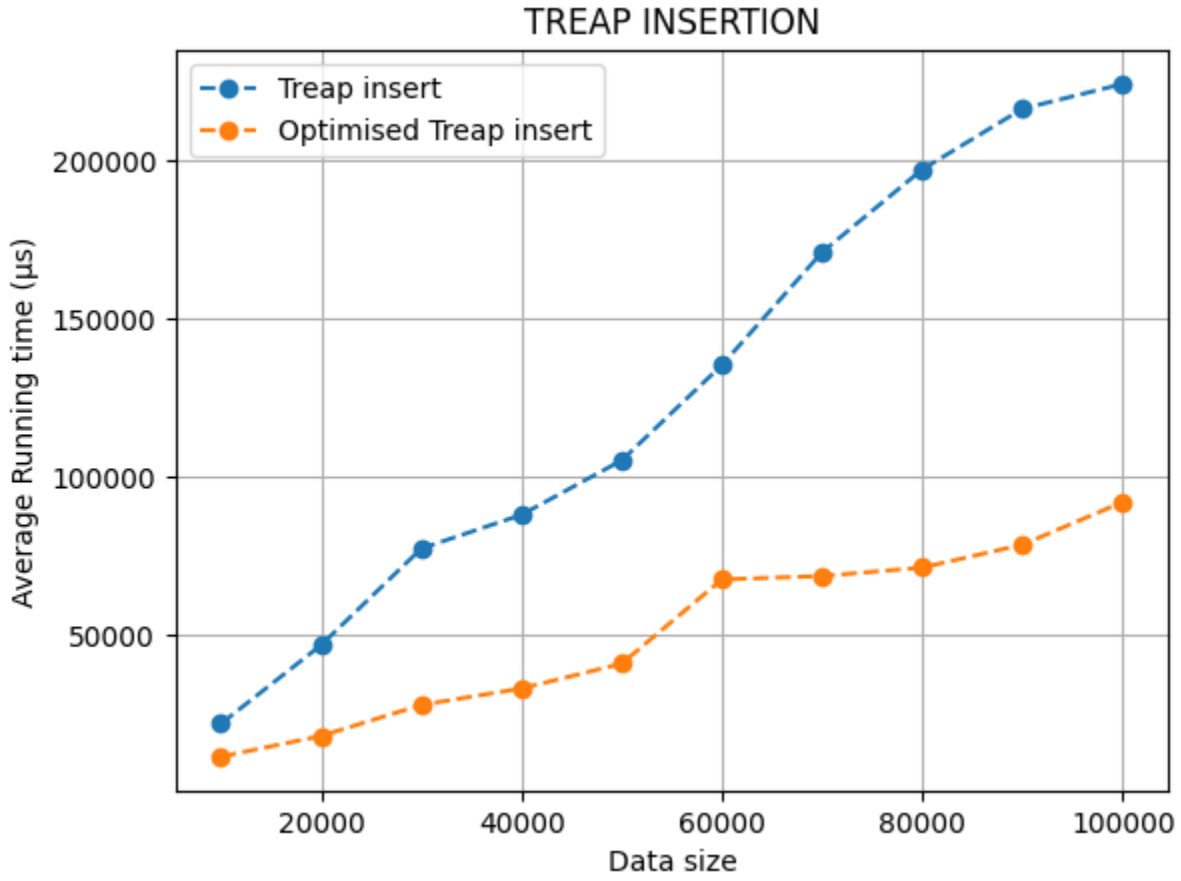
The optimized `insertHelper` function swaps recursion for an iterative approach, introducing a `parent` pointer for efficient traversal. Handling duplicate keys is streamlined, directly appending indices to existing nodes. Node creation is moved outside the loop, and rotations occur within it, ensuring heap property. The return value is adjusted to maintain consistency. These changes collectively result in a more organized and efficient insertion process for the Treap data structure.

Note: Since the optimized code is too large for a snippet, the link is given here.

4.2.3 Changes and Impact

The optimized insertion function utilizes an iterative traversal approach, maintaining O(log n) time complexity on average. It separates duplicate key handling from traversal and delays tree rotations until after insertion. These changes streamline operations, potentially improving efficiency, particularly for large datasets. Overall, the optimized function enhances time complexity through iterative traversal and streamlined operations while maintaining steady memory complexity.

4.2.4 Testing and Validation



Comparison and Conclusion

Determining the "best" data structure often depends on the specific requirements of your application. Each data structure, including Binary Search Trees (BSTs), AVL trees, and Treaps, has its own advantages and disadvantages.

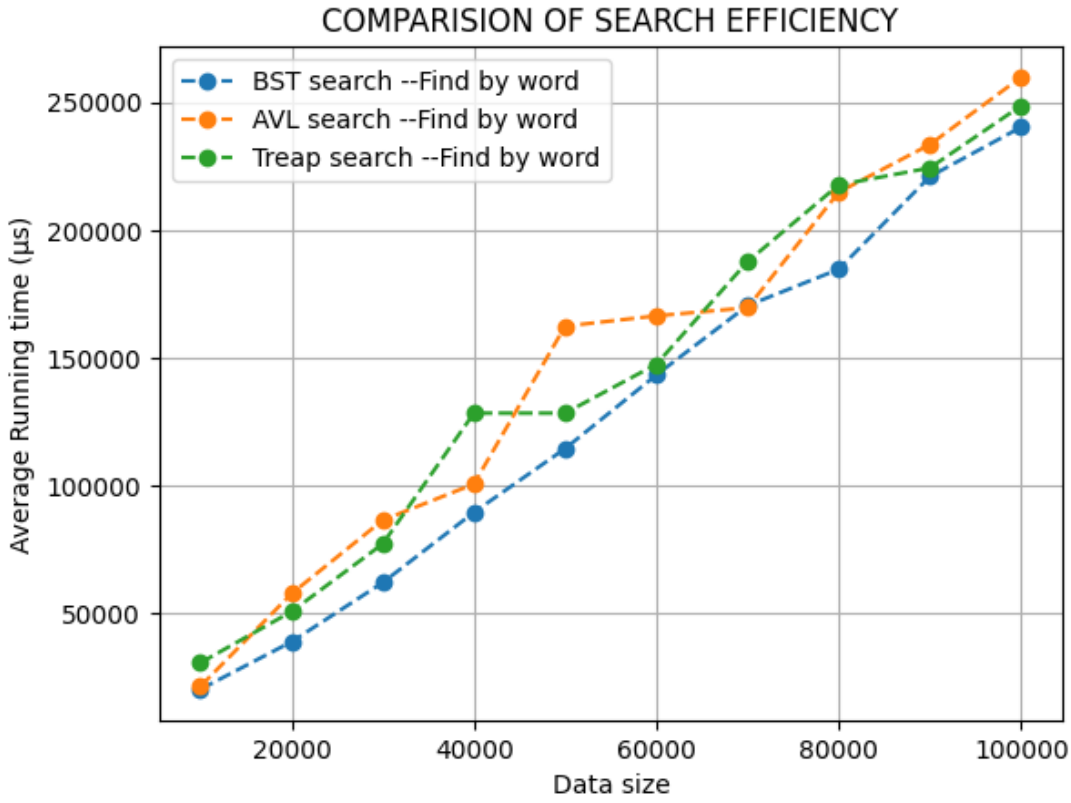
1. BST (Binary Search Tree): This is a fundamental data structure with relatively simple insertion, deletion, and search operations. However, the performance of BSTs can degrade if the tree becomes unbalanced, leading to worst-case time complexities. Its worst-case time complexity in O(n) for insertion, search, and deletion.
2. AVL Tree: AVL trees are self-balancing binary search trees. They maintain a balance factor for each node, ensuring that the tree remains balanced after insertions and deletions. This guarantees that operations like search, insertion, and deletion have a worst-case time complexity of O(log n), making AVL trees suitable for applications where predictable performance is crucial.
3. Treap: A Treap is a combination of a binary search tree and a heap. In a Treap, each node has both a key (like in a BST) and a priority (like in a heap). The priorities follow the heap property, meaning the priority of a node is greater than or equal to the priorities of its children. The keys follow the binary search tree property, meaning keys in the left subtree are less than the key in the node, and keys in the right subtree are greater. Treaps offer the benefits of both BSTs and heaps. They maintain the order of a BST while ensuring relatively balanced trees through the heap priorities. Because of this randomized balancing technique, Treaps have a good expected performance for insertion, deletion, and search operations, with an average-case time complexity of O(log n). However, their worst-case time complexity can be O(n) if the priorities are not well distributed.

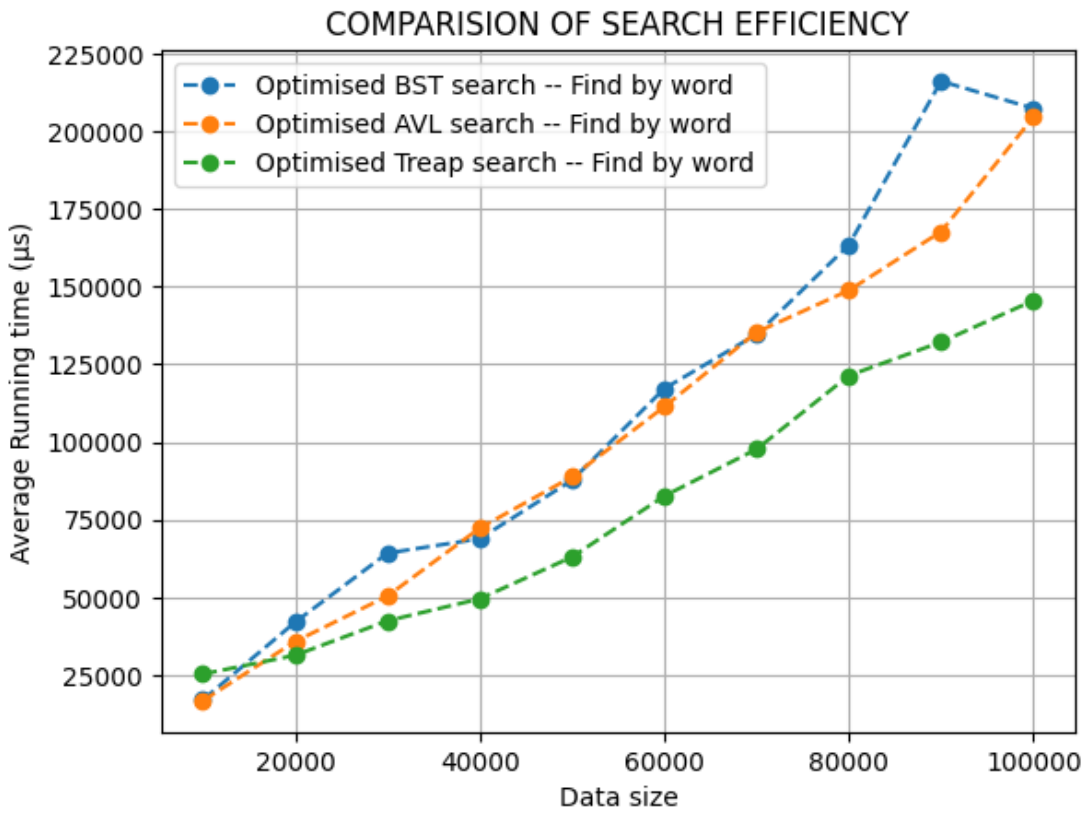
However, Treaps might be considered advantageous because of:

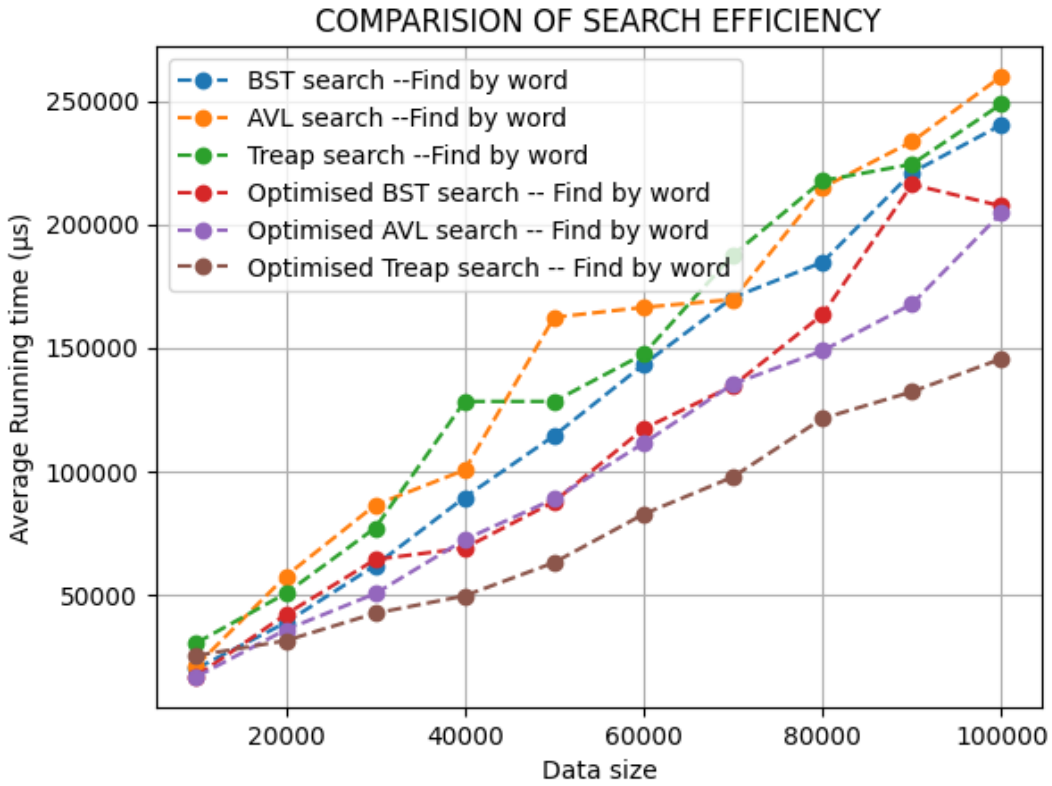
1. Randomized balancing: Treaps use random priorities to balance the tree. This randomness helps prevent worst-case scenarios that might occur with other balancing techniques.
2. Efficiency: In practice, Treaps often perform well due to their average-case time complexities for common operations.
3. Simplicity: Compared to AVL trees, Treaps can be simpler to implement.

Which is why our optimized version of Treaps performs the best as compared to AVL and BST.

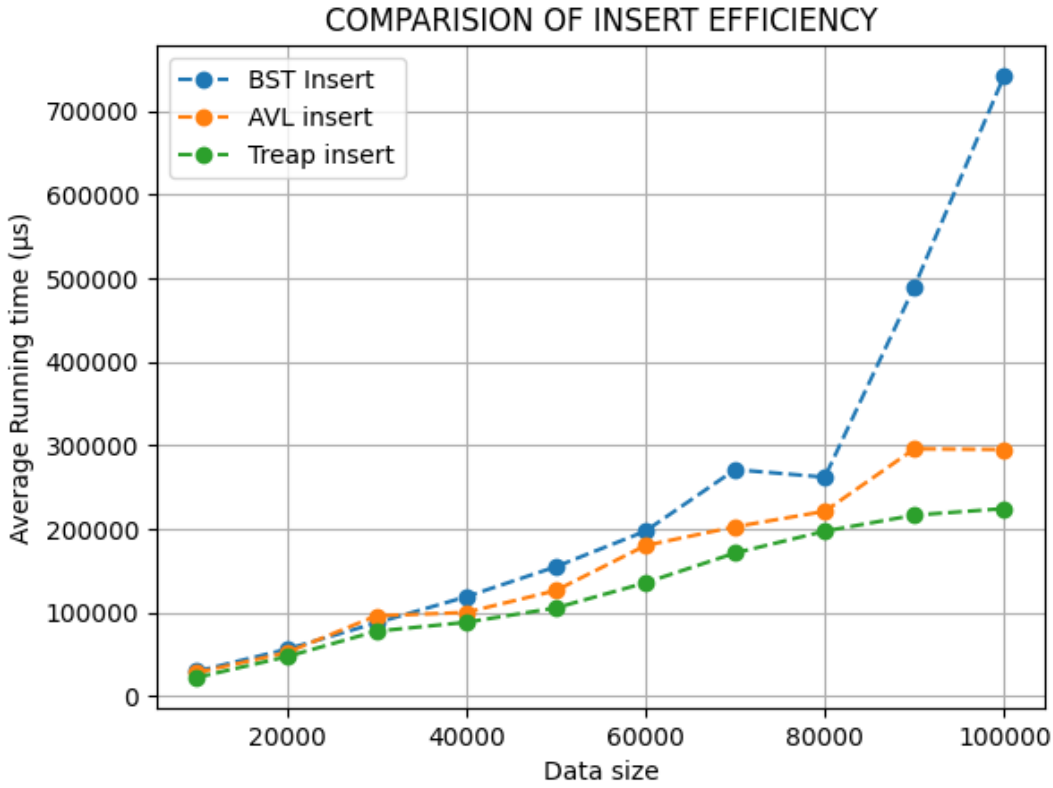
Search


COMPARISION OF SEARCH EFFICIENCY
BST search --Find by word
AVL search --Find by word
Treap search --Find by word
Average Running time (µs)
250000
200000
150000
100000
50000
20000
40000
60000
80000
100000
Data size

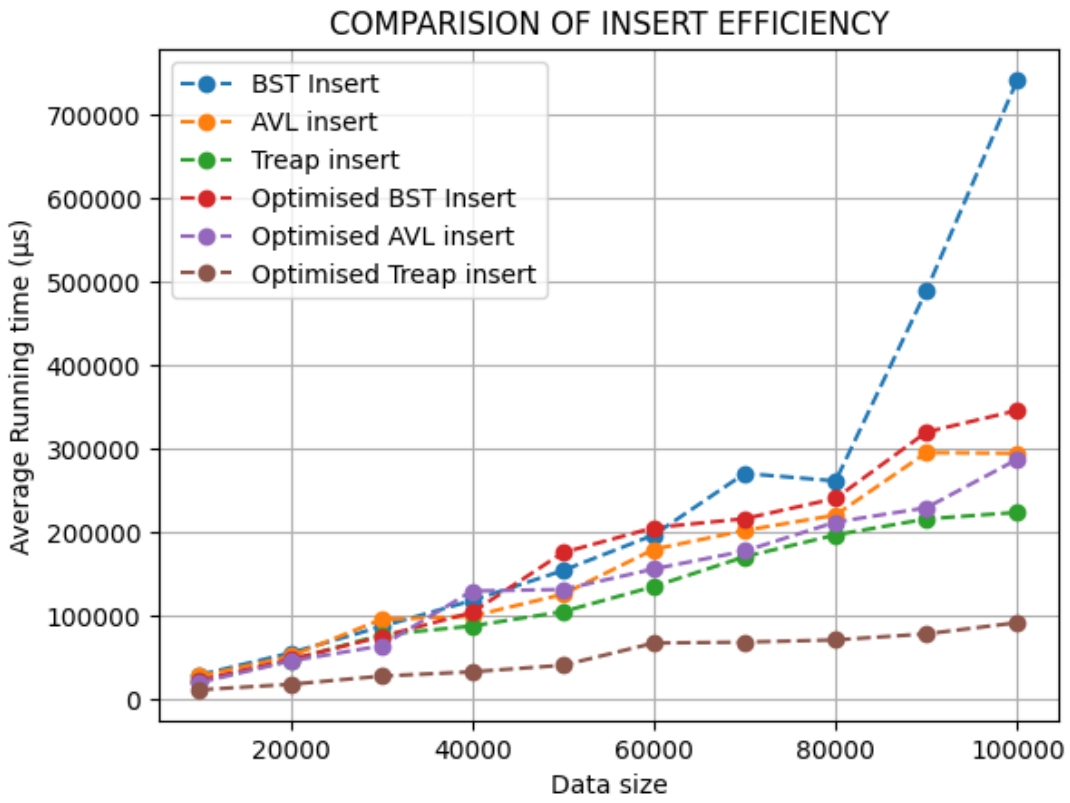

COMPARISION OF SEARCH EFFICIENCY
Optimised BST search -- Find by word
Optimised AVL search -- Find by word
Optimised Treap search -- Find by word
Average Running time (µs)
Data size
225000
200000
175000
150000
125000
100000
75000
50000
25000
20000
40000
60000
80000
100000


COMPARISION OF SEARCH EFFICIENCY
BST search --Find by word
AVL search --Find by word
Treap search --Find by word
Optimised BST search -- Find by word
Optimised AVL search -- Find by word
Optimised Treap search -- Find by word
Average Running time (µs)
Data size
250000
200000
150000
100000
50000
20000
40000
60000
80000
100000

Insertion


COMPARISION OF INSERT EFFICIENCY
BST Insert
AVL insert
Treap insert
Average Running time (µs)
0
100000
200000
300000
400000
500000
600000
700000
20000
40000
60000
80000
100000
Data size


COMPARISION OF INSERTION EFFICIENCY
Optimised BST Insert
Optimised AVL insert
Optimised Treap insert
Average Running time (µs)
0
50000
100000
150000
200000
250000
300000
350000
20000
40000
60000
80000
100000
Data size


COMPARISION OF INSERT EFFICIENCY
BST Insert
AVL insert
Treap insert
Optimised BST Insert
Optimised AVL insert
Optimised Treap insert
Average Running time (µs)
0
100000
200000
300000
400000
500000
600000
700000
20000
40000
60000
80000
100000
Data size

Time Analysis

Steady clock, system clock, and high-resolution clock[1] are often used in computer science and programming to manage timekeeping and accuracy. While the system clock is synchronized with external sources, it can be adjusted by the operating system, leading to potential inaccuracies. High-resolution clocks offer finer granularity but lack stability. In contrast, a steady clock provides a reliable and stable time reference. Its consistency ensures accuracy, making it preferable when timing precision is important. We tested our program with both high resolution and steady clocks, finding that while high resolution introduced frequent and larger inconsistencies, the steady clock provided consistent and reliable time measurements with rare anomalies. Hence, we used the **std::chrono::steady_clock()** to test how long it took for our programs to run. To ensure accuracy, we conducted ten iterations of each program on the same machine. In each iteration, we used different values, such as words for searching and inserting, while maintaining consistency across each dataset. By averaging the running times across these iterations, we aimed to obtain the most precise values possible.

BST																						
	findByWord											Insertion										
	1st Try	2nd Try	3rd Try	4th Try	5th Try	6th Try	7th Try	8th Try	9th Try	10th Try	Average	1st Try	2nd Try	3rd Try	4th Try	5th Try	6th Try	7th Try	8th Try	9th Try	10th Try	Average
dataSet1 (10k)	23270	20108	27176	18660	16230	17011	21673	17666	19737	18661	20019.2	34956	44255	43083	35021	26686	22589	25778	22451	25968	15011	29579.8
dataSet2 (20k)	38879	38785	36717	44715	37590	43437	33263	35783	39918	37421	38650.8	43109	37149	39012	48531	59110	41339	45149	57113	122819	65420	55875.1
dataSet3 (30k)	63392	62559	58751	57817	53730	59004	51996	50516	71027	89344	61813.6	80140	66574	62968	106898	133145	96380	60894	59314	69485	138514	87431.2
dataSet4 (40k)	143031	77754	82639	95562	76457	87989	77184	81286	89630	82239	89377.1	142739	106999	118270	117448	150070	108145	81762	114021	115927	131334	118671.5
dataSet5 (50k)	142760	137679	116991	117639	104993	102941	99040	110176	116863	94293	114337.5	174525	115410	128795	177731	136629	114889	154167	144387	155614	243800	154594.7
dataSet6 (60k)	151120	122042	148353	156932	109396	137955	139963	159296	177644	129044	143174.5	221917	258610	166730	181586	144050	145535	176712	231489	240542	205157	197232.8
dataSet7 (70k)	188336	170630	164194	154153	174208	214910	149163	140184	157666	189935	170337.9	262326	223766	316261	392389	319312	253293	233035	313099	214827	180231	270853.9
dataSet8 (80k)	183952	181835	179568	180626	223070	198602	193638	152770	168316	183622	184599.9	337037	216088	201190	221172	314169	243876	201900	287360	231885	363754	261843.1
dataSet9 (90k)	212823	201374	225661	235504	237565	234629	211645	200596	229161	220368	220932.6	329138	226664	253834	335606	575980	649739	650407	608267	608750	662415	490080
dataSet10 (100k)	255362	245276	240742	242577	212379	232384	235753	266278	239825	232141	240271.7	679597	696667	760738	700338	738175	788586	827303	728445	787467	714699	742201.5

BST (OPTIMIZED VERSION)																						
	findByWord											Insertion										
	1st Try	2nd Try	3rd Try	4th Try	5th Try	6th Try	7th Try	8th Try	9th Try	10th Try	Average	1st Try	2nd Try	3rd Try	4th Try	5th Try	6th Try	7th Try	8th Try	9th Try	10th Try	Average
dataSet1 (10k)	18500	14681	14768	14930	17010	23193	16844	15156	14464	20267	16981.3	31322	32054	22867	20789	14210	22711	16828	26448	24487	19201	23091.7
dataSet2 (20k)	41719	50208	34627	35818	29955	30683	45604	47336	33439	70413	41980.2	40773	44618	50626	57901	37787	41219	66276	41233	45601	54080	48011.4
dataSet3 (30k)	78316	53344	56380	51220	74527	94446	54145	50853	74964	53802	64199.7	63790	100755	86757	84125	76723	65463	50627	60576	75999	89088	75390.3
dataSet4 (40k)	66136	68759	60877	61185	65699	60889	62481	100778	64253	77415	68847.2	69618	124935	149642	87770	115213	91281	99886	74971	96964	132976	104325.6
dataSet5 (50k)	93666	92915	89712	94380	82636	83695	79625	78740	78633	102398	87640	179443	141373	140283	117929	249171	180343	226669	153423	237653	134215	176050.2
dataSet6 (60k)	111872	121508	180961	113148	96861	112771	100785	103700	108705	121128	117143.9	303521	178675	274488	146484	185792	234167	136688	118943	221962	257972	205869.2
dataSet7 (70k)	125961	135295	141434	120007	141309	139753	121848	119322	157230	143866	134602.5	236939	313553	259998	171136	255448	255932	184910	163694	141938	182679	216622.7
dataSet8 (80k)	136575	136854	200961	163274	134520	173393	136980	172422	162646	215926	163355.1	296635	166678	206753	304859	226275	214930	338435	248158	176981	229726	240943
dataSet9 (90k)	287223	191377	195621	209404	206577	202339	300523	177349	222445	167994	216085.2	317762	344878	335447	280617	319262	289328	371705	376670	294886	273332	320388.7
dataSet10 (100k)	207002	184204	183830	220183	183270	191711	220001	208730	251492	223188	207361.1	386770	297371	273971	406457	360695	362054	393426	300322	374778	306891	346273.5

Fig 5.1 Time Analysis (BST vs Optimized)

[1] https://cplusplus.com/forum/general/187899/

https://stackoverflow.com/questions/13263277/difference-between-stdsystem-clock-and-stdsteady-clock

Challenges

1. The OS Scheduler[2] determines how to move processes between the ready and run queries which can only have one entry per processor core on the system. Due to which we came across anomalies in time while testing our programs, when we repeatedly and consecutively tested many values.
2. We had to change datasets repeatedly to ensure very large datasets with even intervals.

Work Division

1. Muneeba: Treaps Original, Treaps Optimized, and Time Analysis
2. Samah: BST Original, Time Analysis, Debugging all 3 data structures
3. Fakeha: AVL Original, BST Optimized, and Report
4. Eman: Treaps Original, Datasets, and Graphs
5. Qurba: AVL Original, AVL Optimized, and Report

[2] https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm#:~:text=The%20OS%20can%20use%20different,been%20merged%20with%20the%20CPU

https://stackoverflow.com/questions/54271755/why-the-execution-time-of-the-same-code-on-the-same-computer-could-be-different

Appendix

Find our project code, and time analysis document here.