
Programming Assignment

Name-Akshansh Yadav

Roll Number- D23CSE001

Problem-1 Title: Efficient File System Indexing: Performance Analysis

Introduction: Rutvik, a software engineer, is leading a pivotal project aimed at optimizing the file system of NextOS, a cutting-edge operating system recognized for its prowess in managing extensive data volumes and enabling swift file access. Your role in Rutvik's team is significant as you collectively tackle the challenge of refining file and directory storage and retrieval. The current data structure in use offers basic indexing capabilities, but the team is eager to explore the benefits of a more advanced structure that could significantly enhance operations like file insertion, searching, and deletion. You've been tasked with conducting a comparative assessment between the existing custom data structure (distinct from a red-black tree) and the red-black tree itself, aiming to determine the most suitable approach for optimizing the file system's performance within NextOS.

Methodology: To facilitate an accurate efficiency comparison, Rutvik employs two distinct data structures: the red-black tree and a hash table utilizing linear probing. The primary focus is on evaluating the time taken for insertion, deletion, and searching operations. The red-black tree is implemented with strict adherence to its fundamental properties, while the hashing approach involves a hash table of size 20011, a prime number. All operations are meticulously executed within these two data structures, ensuring a comprehensive assessment of their respective performance.

Implementation of Red-Black Tree :

The code implements a red-black tree in C for efficient insertion, searching, and deletion of elements. It defines the structure of a node with data, color, left and right children, and parent pointers. Key functions include creating nodes, left and right rotations, fixing the tree after insertions, and deletion. In the main function, integers are read from "input.txt," inserted into the red-black tree, and tree operations (insertion, searching, and deletion) are showcased with predefined arrays. The code illustrates the core functionality of a red-black tree.

Implementing of Hashing :

The Implemented code demonstrates a basic implementation of a hash table using linear probing for collision resolution. It includes functions for insertion, searching, and deletion operations. The program initializes the hash table, reads integer values from a file, inserts them, and showcases insertion, searching, and deletion on predefined value arrays. The hash table uses a simple modulo-based hash function and linear probing for addressing collisions. While the code provides a foundational understanding of hash table operations, it requires comprehensive testing and optimization for real-world applications.

Comparisons of Time Taken for Execution of Red BlackTree and Hashing

To benchmark the insertion time of 10,000 elements, Rutvik is utilizing the built-in C library "<time.h>". This involves measuring the execution time by recording the start and end times using appropriate variables, and then calculating the time difference. Upon comparison, it's observed that the red-black tree implementation takes marginally more time compared to the hash table solution.

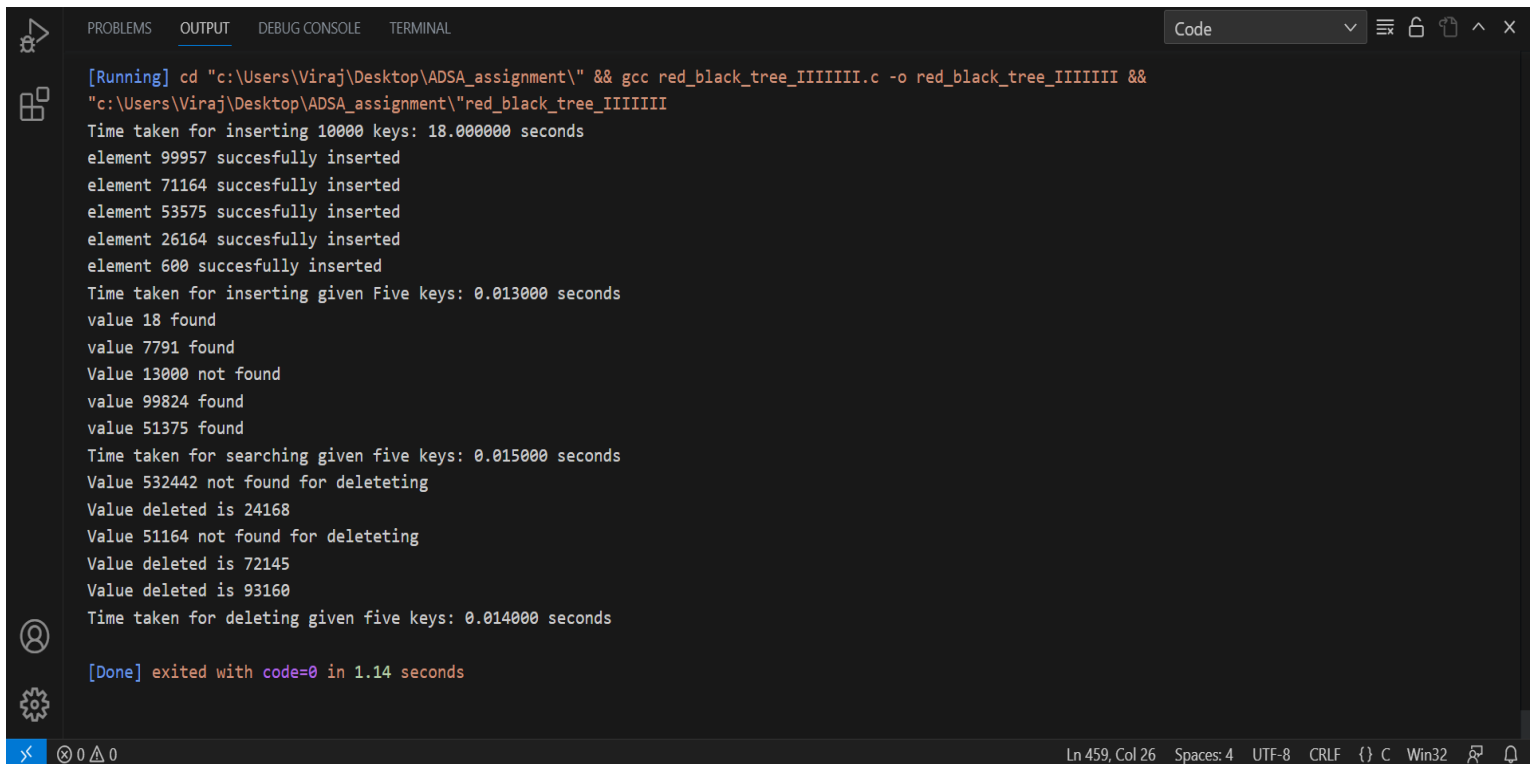
Sample Inputs For Comparisons:

Element to insert: 99957, 71164, 53575, 26164, 600

Element to search: 18, 7791, 13000, 99824, 51375

Element to Delete: 51, 24168, 51164, 72145, 93160

Result Output of Red Black Tree Data Structure :



```
[Running] cd "c:\Users\Viraj\Desktop\ADSA_assignment\" && gcc red_black_tree_IIIIIII.c -o red_black_tree_IIIIIII &&
"c:\Users\Viraj\Desktop\ADSA_assignment\"red_black_tree_IIIIIII
Time taken for inserting 10000 keys: 18.000000 seconds
element 99957 succesfully inserted
element 71164 succesfully inserted
element 53575 succesfully inserted
element 26164 succesfully inserted
element 600 succesfully inserted
Time taken for inserting given Five keys: 0.013000 seconds
value 18 found
value 7791 found
Value 13000 not found
value 99824 found
value 51375 found
Time taken for searching given five keys: 0.015000 seconds
Value 532442 not found for deleteting
Value deleted is 24168
Value 51164 not found for deleteting
Value deleted is 72145
Value deleted is 93160
Time taken for deleting given five keys: 0.014000 seconds

[Done] exited with code=0 in 1.14 seconds
```

Key Points to be Noted:

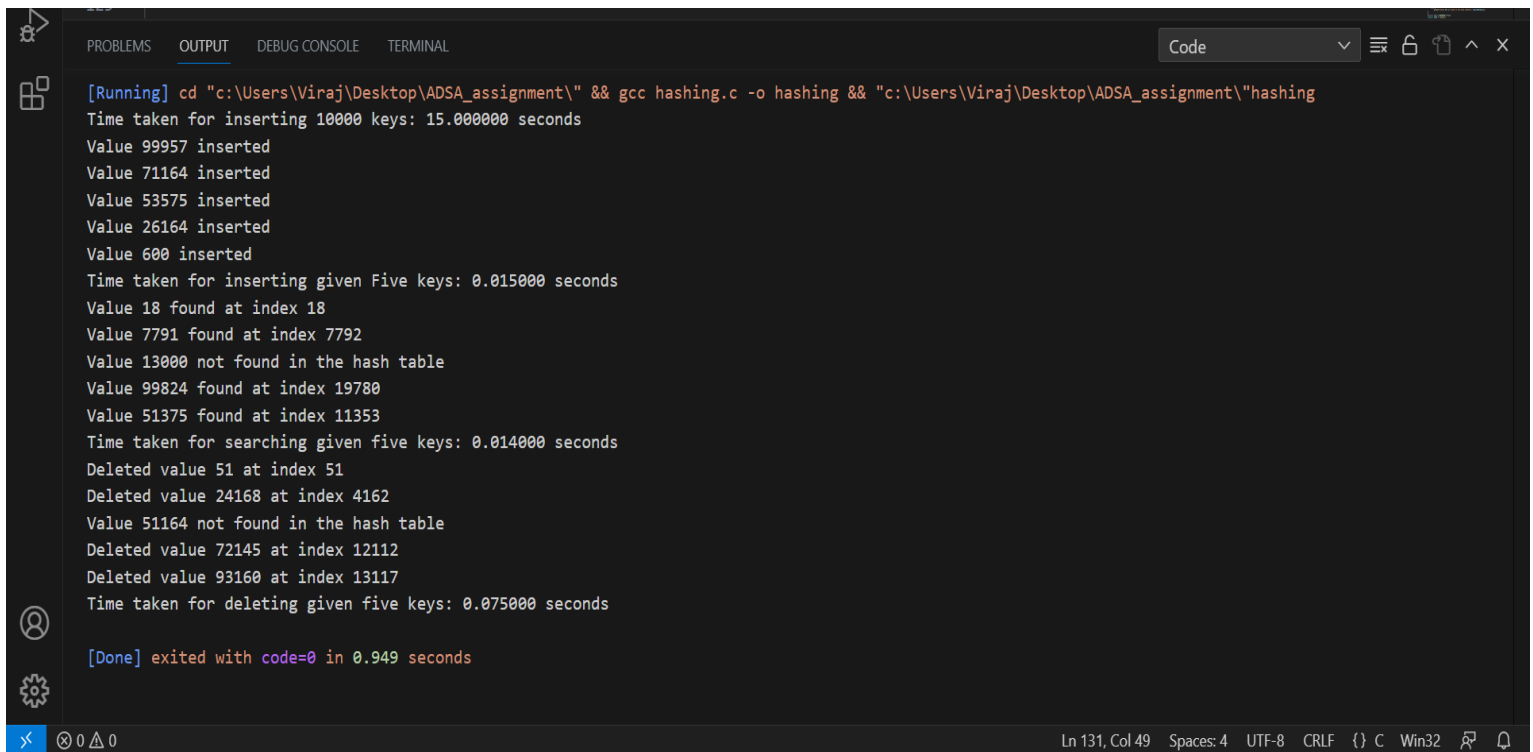
It is cleary shown that the time taken to insert 10000 element in red black tree is 18.000 seconds.

In Order to insert the Five keys in the given red black tree it is taking 0.013 seconds.

In Searching the Given Five Keys in red black tree it is taking 0.015 seconds.

In deleting the given five keys red black tree is taking 0.014 seconds.

Result Output of Hashing Data Structure :



```
[Running] cd "c:\Users\Viraj\Desktop\ADSA_assignment\" && gcc hashing.c -o hashing && "c:\Users\Viraj\Desktop\ADSA_assignment\"hashing
Time taken for inserting 10000 keys: 15.000000 seconds
Value 99957 inserted
Value 71164 inserted
Value 53575 inserted
Value 26164 inserted
Value 600 inserted
Time taken for inserting given Five keys: 0.015000 seconds
Value 18 found at index 18
Value 7791 found at index 7792
Value 13000 not found in the hash table
Value 99824 found at index 19780
Value 51375 found at index 11353
Time taken for searching given five keys: 0.014000 seconds
Deleted value 51 at index 51
Deleted value 24168 at index 4162
Value 51164 not found in the hash table
Deleted value 72145 at index 12112
Deleted value 93160 at index 13117
Time taken for deleting given five keys: 0.075000 seconds

[Done] exited with code=0 in 0.949 seconds
```

Key Points to be Noted:

It is clearly shown that the time taken to insert 10000 element in hash table is 15.000 seconds

In Order to insert the Five keys in the given hash table it is taking 0.015 seconds.

In Searching the Given Five Keys in hashing table it is taking 0.014 seconds.

In deleting the given five keys hashing is taking 0.0750 seconds.

Conclusion:

Ease and Efficiency of Creation: The creation of a hash table is notably simpler and faster compared to constructing a red-black tree.

Performance of Operations: In the red-black tree, fundamental operations such as insertion, deletion, and searching tend to be more time-intensive in comparison to hashing.

Indexing Efficiency: For indexing purposes, hashing emerges as the preferred choice over red-black trees due to its more efficient and expedient nature.

This analysis underscores the advantages of employing hashing, particularly in scenarios where indexing tasks are prevalent.

Problem 2 : Convert an arbitrary n-node binary search tree to a given n-node binary search tree.

Introduction :

Converting an arbitrary n-node binary search tree into a specified n-node binary search tree involves rearranging the tree's structure and values while preserving the binary search property. This process aims to align each node according to its ordering within the new tree, showcasing the dynamic adaptability of binary search trees.

Approach:

Node Creation: The `create_node` function generates tree nodes.

Original Tree Display: `inorder_traversal` displays the original tree's elements.

Value Replacement: `inorder_traversal_REPLACE` replaces tree values with array values using in-order traversal.

Pre-order Display: `pre_order_traversal` shows the modified tree elements in pre-order.

Main Process: The program initializes a placeholder tree structure. It replaces values using the array while retaining the original layout. The pre-order and in-order traversals of the modified tree are displayed.

Implementation:

We are considering that the given Binary Search Tree is given in the form of Inorder traversal 5, 10, 12, 20, 22, 25, 28, 30, 32, 36, 40, 64

In order to maintain the structure of given Binary Search Tree, Our approach is to maintain the structure by creating a dummy tree.

Insert nodes in the tree in the form of Inorder Traversal, In doing so we are able to maintain the property of red black tree and also the given structure.

Program Output of Binary Search Tree

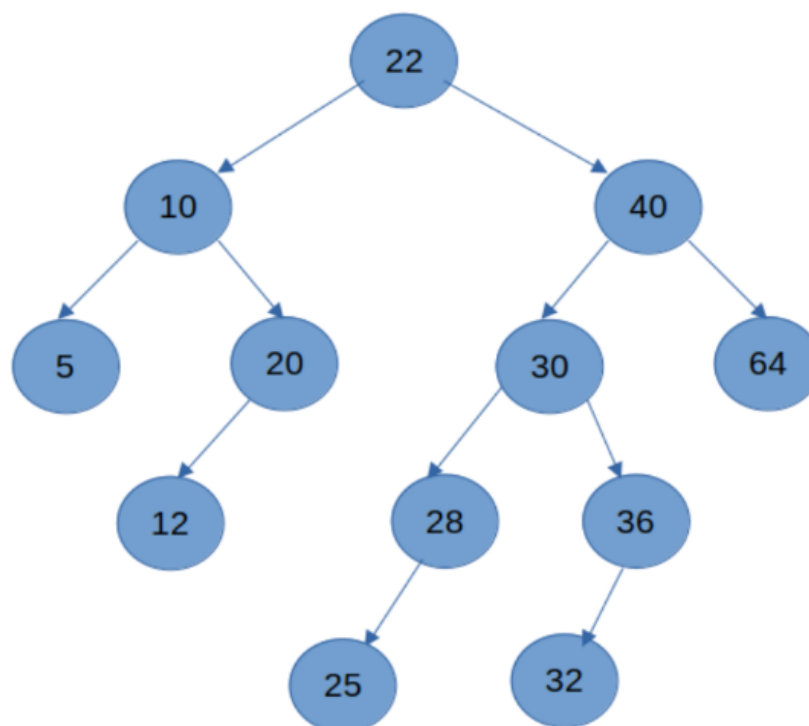
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[Running] cd "c:\Users\Viraj\Desktop\ADSA_assignment\" && gcc Answer2.ADSA.c -o Answer2.ADSA && "c:\Users\Viraj\Desktop\ADSA_assignment\"Answer2.ADSA
Structure of the given tree is maintained
Pre_order Traversal of Tree of Required Structure: 22 10 5 20 12 40 30 28 25 36 32 64
In_order Traversal of Tree of Required Structure: 5 10 12 20 22 25 28 30 32 36 40 64
[Done] exited with code=0 in 2.209 seconds
```

With the help of the pre_order Traversal and In_order traversal we can create a unique Binary Search Tree

Pre_order Traversal of Tree of Required Structure: 22 10 5 20 12 40 30 28 25 36 32 64

In_order Traversal of Tree of Required Structure: 5 10 12 20 22 25 28 30 32 36 40 64

Unique Binary Search Tree structure is-



Problem 3 : implement Dijkstra's algorithm using Red Black Tree.

Introduction :

Dijkstra's algorithm is a well-known method for discovering the shortest paths in weighted graphs. It has broad applications in network routing and transportation systems. This implementation explores an intriguing adaptation, utilizing the red-black tree structure to efficiently manage the algorithm's priority queue. This amalgamation highlights the versatility of red-black trees in optimizing Dijkstra's algorithm execution, demonstrating their prowess in handling graph-related challenges.

Implementation Approach :

In order to implement dijkstra using red black tree we are using multiple functions in C

```
node * create_node(int vertex,int distance)

void traverse_inorder(node* temp);

node * BST(node * trav, node* temp);

void traverse_preorder(node * temp);

void left_rotate(node * temp);

void right_rotate(node* temp);

void fixing(node* root, node * ptr);

void red_black_transplant(struct node *u, struct node *v);

void red_black_delete_fixup(struct node *x);

node *tree_search(int key);

node *tree_minimum(struct node *x);

void red_black_delete(struct node *z);

node* findMin(node* root);

node* extractMin(node** root);

void dijkstra(int matrix[N][N], int source);
```

Functionality Description :

create_node(int vertex, int distance): Creates a new node with the specified vertex and distance.

BST(node *trav, node *temp): Performs binary search tree insertion while maintaining the red-black tree properties.

fixing(node *root, node *ptr): Fixes the red-black tree properties after an insertion or distance update.

red_black_transplant(node *u, node *v): Replaces a subtree rooted at node u with a subtree rooted at node v in the red-black tree.

tree_search(int key): Searches for a node with the specified key (vertex value) in the red-black tree.

tree_minimum(node *x): Finds the node with the minimum distance in the red-black tree.

red_black_delete(struct node *z): Deletes a node z from the red-black tree and maintains the red-black properties.

findMin(node *root): Finds the node with the minimum distance in the red-black tree.

extractMin(node **root): Extracts and deletes the node with the minimum distance from the red-black tree.

dijkstra(int matrix[N][N], int source): Implements Dijkstra's algorithm using a red-black tree as a priority queue.

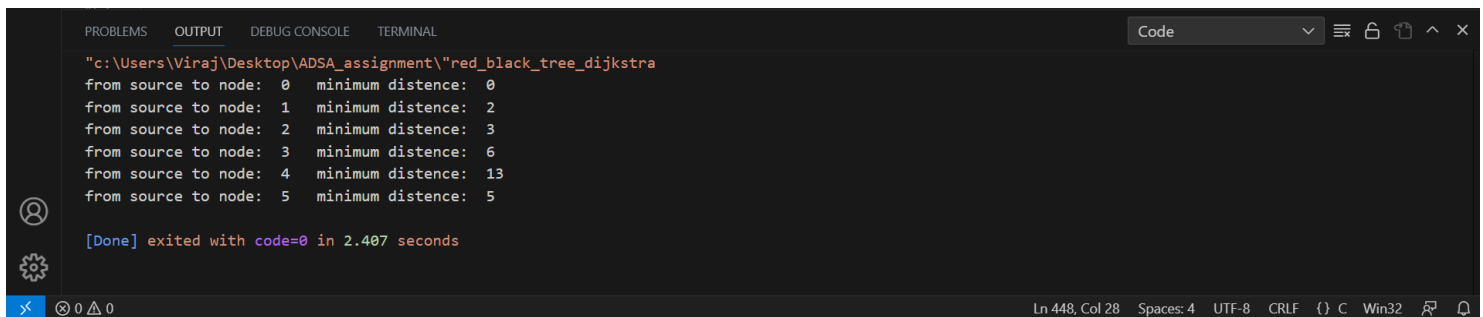
main(): Initializes the adjacency matrix, sets the source vertex, and invokes the **dijkstra** function to compute the shortest distances from the source vertex to all other vertices.

The red-black tree is used to efficiently maintain the vertex-distance pairs, perform updates, and extract the vertex with the minimum distance during each iteration of Dijkstra's algorithm. This approach improves the time complexity of selecting the next vertex and updating distances compared to traditional linear priority queues.

Given Input Matrix :

```
int matrix[N][N] = {  
  
    {0, 2, 3, 0, 0, 5},  
  
    {2, 0, 0, 4, 0, 0},  
  
    {3, 0, 0, 0, 0, 6},  
  
    {0, 4, 0, 0, 7, 0},  
  
    {0, 0, 0, 7, 0, 8},  
  
};
```

Result Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
Code  
"c:\Users\Viraj\Desktop\ADSA_assignment\red_black_tree_dijkstra  
from source to node: 0    minimum distance: 0  
from source to node: 1    minimum distance: 2  
from source to node: 2    minimum distance: 3  
from source to node: 3    minimum distance: 6  
from source to node: 4    minimum distance: 13  
from source to node: 5    minimum distance: 5  
  
[Done] exited with code=0 in 2.407 seconds  
Ln 448, Col 28  Spaces: 4  UTF-8  CRLF  {}  C  Win32
```

The output of the program displays the minimum distances from the specified source vertex to all other vertices in the given graph. Each line in the output corresponds to a vertex and its corresponding minimum distance.

Combining these complexities, the overall time complexity of the Dijkstra's algorithm with Red-Black Tree implementation is dominated by the Red-Black Tree operations, which is $O(N \log N)$ due to the insertion, deletion, and Red-Black Tree fixing steps.

Conclusion :

The Red-Black Tree's operations of insertion, deletion, and fixing ensure that the algorithm maintains its overall time complexity of $O(N \log N)$ for dense graphs, making it a viable alternative to other data structures like priority queues. However, it's important to note that in practice, for very large graphs, specialized data structures like Fibonacci Heaps might provide better performance due to their lower constant factors and amortized time complexity.

This implementation showcases the flexibility and adaptability of various data structures in solving algorithmic problems. The combination of Dijkstra's algorithm and the Red-Black Tree data structure provides an efficient solution for finding shortest paths, making it a valuable addition to the toolbox of algorithms and data structures for graph-based applications.