Programming Project

Project 2 (Search Algorithms)

INTRODUCTION:

Search algorithms are a very important part of computer science. They serve as a foundation for many applications. Search algorithms are used for quick data retrieval and to find duplicates in the data structure. These algorithms are very useful in systems like information retrieval, database administrator, AI etc. Understanding the search algorithms and optimization techniques helps lower the processing cost and helps achieve results faster.

Objective of the project:

This project aims to compare and evaluate the runtime performance of four different search algorithms: Linear Search, Binary Search (sorted array), Binary Search Tree, and Red-Black Tree. This information can be helpful to know about the efficiency of each algorithm, helping gain insight into their use case scenarios, helping in making right algorithm selection in right situations.

Overview of the Algorithms:

1. Linear Search:

Linear search is a straightforward search algorithm that checks every element in the data structure one after the other from left to right. The run time complexity of this algorithm is O(n), meaning that the execution time grows linearly with the size of the input.

2. Binary Search (within a sorted array):

Binary Search algorithms are more efficient algorithms which work best with sorted inputs. It

Works by separating the array into 2 halves every iteration until the key element that we are searching for is found or the interval is empty. The time complexity of binary search is O (log n), which means that the run time grows logarithmic and is better than O(n).
Binary search needs a sorted array to work but is faster than linear search for large data sets.

3. Binary Search Tree (BST):

Binary search tree is a tree data structure that always maintains a simple rule to keep all the smaller elements to left and the larger elements to right. This rule is followed at all the nodes. This

way on each iteration of the search half of the tree is ignored. The time complexity of BST is O (log n) on average. However when the tree is Skewed the time complexity degrades to O(n) similar to Linear search.

4. Red-Black Tree:

A red black tree is a binary tree that self-balances the tree to avoid the worst case in BST, by making sure that the tree always remains approximately balances. This is achieved by a condition called color balance. The time complexity of the red black tree is O (log n) and the self-balancing property helps to maintain the logarithmic time complexity, making it suitable for applications with huge insertion and deletions operations.

Project Structure:

main.py:

The 'main.py' file acts as the project's graphical user interface (GUI), allowing users to interact with the built search algorithms. This GUI, built with the CustomTkinter module, is intended to be simple and useful. It has three separate buttons, each with a special purpose:

1. **Run the GUI:** This button launches a GUI, allowing the users to interact with the input variables and the search algorithms.
2) **RUN CLI:** This button lets user use a CLI interface to interact with the project. This is helpful when large input sizes are used.
3. **Compare with graphs:** This button initiates a comparison of the search algorithms' runtimes, with the results shown in graphical form providing a visual depiction of their performance.

The subprocess module is used to run the scripts associated with each button, ensuring a smooth transition between the GUI, CLI, and graph comparison features.

search_algorithms.py

The “search_algorithms.py” has the implementation of all for algorithms. These functions are imported into other GUI files to implement the algorithms,

Below is a brief explanation of each function, its Arguments and expected returns.

linear_search(arr, key):
Perform linear search to find the index of the key in the array.
Args:
- arr (list): The input array to search.
- key (int): The key to search for.
Returns:
- int: Index of the key if found, otherwise -1.

binary_search_in_sorted_array(arr, key):
Perform binary search on a sorted array to find the index of the key.
Args:
- arr (list): The sorted input array to search.
- key (int): The key to search for.
Returns:
- int: Index of the key if found, otherwise -1.

insert_into_BST(root, key):
Insert a key into Binary Search Tree.
Args:
- root (Node): The root node of the BST.
- key (int): The key to insert.
Returns:
- Node: The root node after insertion.

search_binary_search_tree(root, key):
Search for a key in Binary Search Tree.
Args:
- root (Node): The root node of the BST.
- key (int): The key to search for.
Returns:
- Node or int: Node containing the key if found, otherwise -1.

search_RB_tree(root, key):
Search for a key in Red-Black Tree.
Args:
- root (Node): The root node of the Red-Black Tree.
- key (int): The key to search for.
Returns:
- Node or int: Node containing the key if found, otherwise -1.

build_RBtree(array):
Build a Red-Black Tree from an array of values.
Args:
- array (list): The input array of values to insert into the Red-Black Tree.
Returns:
- Node: The root node of the constructed Red-Black Tree.

CLI.py:

The `CLI.py` script provides a Command Line Interface (CLI) for users to interact with the implemented search algorithms directly from the terminal. It offers a text-based interface,

allowing users to execute the search algorithms, input data, and retrieve results efficiently. This is recommended for very large sized inputs.

Invocation:

The `CLI.py` script can be invoked from the terminal using the command `python CLI.py`. Once launched, users can follow the on-screen prompts to perform various operations, such as searching for elements using different algorithms.

graph_comparison.py:

The `graph_comparison.py` script is designed to visualize and compare the runtime performance of the implemented search algorithms using graphical representations. It generates bar charts or line graphs to showcase the execution times of each algorithm for different input sizes.

Data Produced:

This script produces graphical representations that display the runtime of each search algorithm for varying input sizes. The data visualized helps in understanding the comparative efficiency of the algorithms under different scenarios.

Visualization:

The results are visualized using libraries like Matplotlib, displaying the algorithm names on the x-axis and the corresponding runtime on the y-axis. This visualization aids in easily identifying the most efficient algorithm for specific search tasks.

Project Setup

Follow these instructions to get a local copy of the project and set it up for testing.

Requirements

- Python 3.6 or newer

Installation

1. Clone or download the project to your local machine.

2. Open a terminal and navigate to the project directory.

3. Install the necessary Python packages using the following command:

    “ pip install -r requirements.txt “

Running the tests

1. Open terminal and navigate to the project directory

2. Run the following command to run the tests

python test.py

Results:

We have obtained valuable information about the individual runtime performances of the developed search algorithms after subjecting them to extensive testing. The comparative effectiveness of each algorithm for a range of input sizes is demonstrated by the following results:

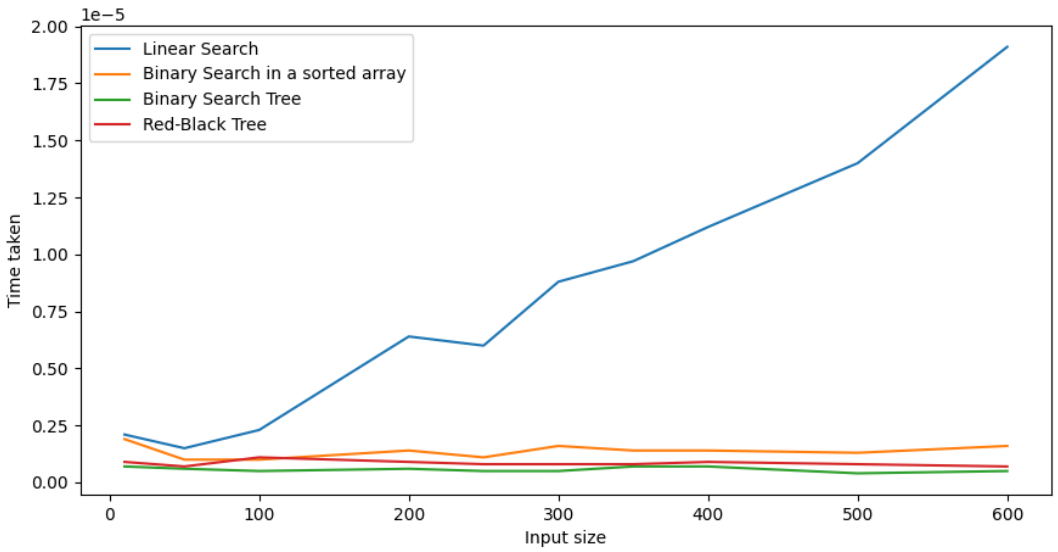

Fig produced using graph_comparison.py.

The visual representation provides a clear understanding of how long each algorithm takes to execute. It makes it easier to understand how quickly and effectively these algorithms search for items across various datasets.

Conclusion:

The runtime comparisons reveal a few noteworthy findings. The logarithmic time complexity of the binary search algorithm indicates that it is more efficient than linear search, especially when used on sorted arrays. For balanced trees, the Binary Search Tree (BST) performs admirably, providing a fair trade-off between time complexity and space requirements. It's interesting to note that the Red-Black Tree exhibits competitive performance in scenarios

where frequent dynamic operations are required, even with its high-cost balancing system.

For small datasets, linear search is a straightforward but effective strategy; however, as dataset sizes increase, its linear time complexity becomes more noticeable and less useful. These results highlight how crucial it is to choose the right algorithm depending on the needs and features of the dataset.

Abhinay Kotla

1002195827