Double Hashing with Binary Search Tree

By:

S.No	Name	Roll No.
1	AdwaitaSathrukkan	CB.EN.U4CSE21403
2	Bapathi Archana	CB.EN.U4CSE21410
3	GollaThanuja	CB.EN.U4CSE21419
4	Harish S	CB.EN.U4CSE21422
5	Sivakami V	CB.EN.U4CSE21456

Introduction:

Hybrid Data Structures have gained popularity due to their unique ability to blend multiple data architectural designs into a singular infrastructure resulting in efficient handling of complex problems. Our project endeavours towards developing such an ambitious concept-based system utilizing Double Hashing and Binary Search Tree(BST) methods aimed at delivering exceptional results when it comes to storing and retrieving large amounts of information effectively.

Double hashing is powerful in resolving conflicts within separately stored hash tables while minimizing clutter. BST organizes every node using individual key values, so it is easy to scan through the chunk of data with precision and without any hassle. The engineering team has blended these techniques together in hybrid architecture, which will offer better search and retrieval features along with faster insertion/deletion options.

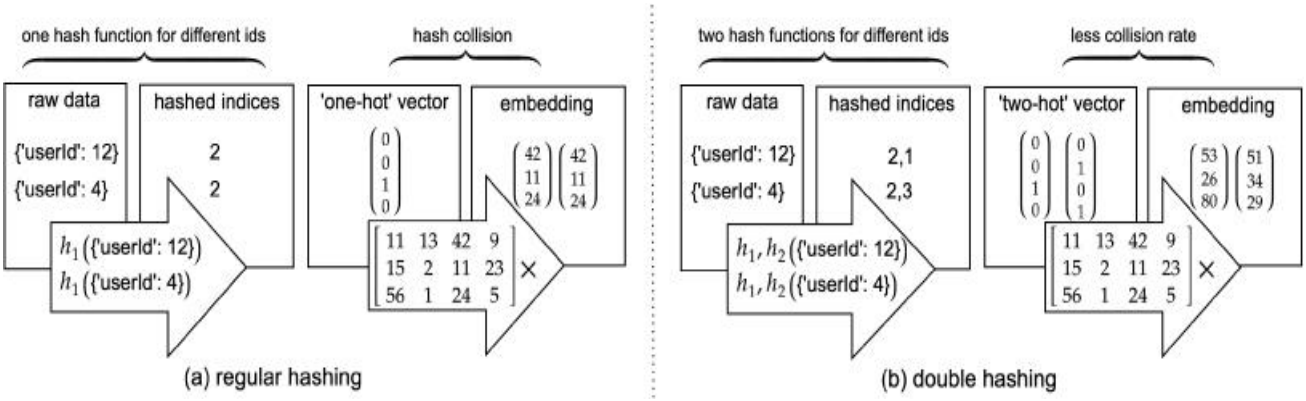
Our hybrid system can be applicable practically in different ventures like managing a high-density database system or organizing folders within the file system to keep sensitive information classified into related batches or ascertaining optimized search results for large volumes of documents within a search engine.

Also, we'll analyze our designed solution thoroughly by measuring its time complexity via executing various operations such as searching, insertion, and deletion under differing input sizes while validating its space complexity to evaluate memory usage at larger scales when compared with popular approaches like hash tables or BSTs. The essence of this project is developing an advanced hybrid data structure by blending Double Hashing with Binary Search Trees' best features effectively. Our motive was to develop a system capable of providing efficient storage capacity alongside fast retrieval and deletion capabilities suitable for multiple applications. Further focus was directed towards analyzing time and space complexity aspects in optimization ideas concerning hybrid structures for improved performance. Ultimately, our goal was to demonstrate the superior qualities of Double Hashing with Binary Search Tree hybrid data structures in efficiently solving complex problems.

Overview of the Hybrid Data Structure:

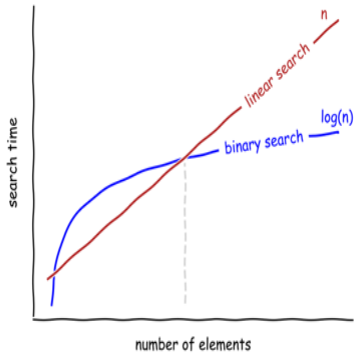
Double Hashing with Binary Search Tree (DH-BST) is a preferred hybrid structure for our project that leverages both techniques, ensuring maximum efficiency in storing as well as retrieving data.

- Double hashing functions as an innovative method that resolves potential collisions stemming from hash tables. By utilizing two hash mechanisms, keys are assigned unique indices within a table on the first occasion. The second mechanism determines probing capabilities of available spaces



whenever a collision occurs, thus minimizing collisions and enhancing overall storage capacity management.

- Equally essential in this hybrid model is Binary Search Trees (BST). We chose BSTs based on their ability to help organize nodes effectively under binary tree structures. Each node operates based on specific principles where the smaller value keys are left in one child node while larger values reside in another to ensure efficient search insertion or deletion tasks according to key orders.

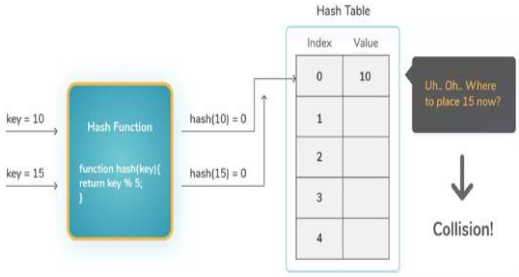


- Our design strategy has involved the fusion of Double Hashing and BST principles. We employed double hashing and utilized effective mapping techniques assigning unique indices for each key accessing hash tables directly until collisions occur when utilizing BST's facet maintaining order within collision chains nodes acting individually like entries maintaining order between them during search insertion or deletion operations needing such chains during use.

Benefits and Purpose:

DH-BST's hybrid structure offers various advantages when dealing with storing or retrieving data swiftly.

- By using double hashing technology, **elements are efficiently** located within the hash table without collision issues since they're appropriately distributed according to the required locations.

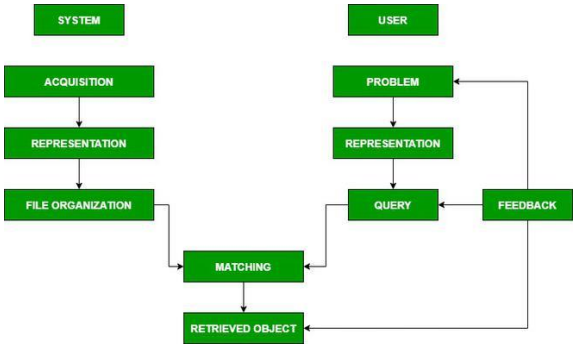


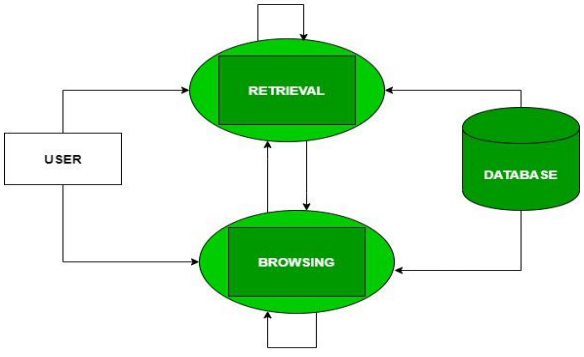
- The binary search tree component ensures that there is no compromise of search speed due to any potential **collisions**. It manages this by preserving key orders within each collision set so you won't have any trouble retrieving these critical pieces of information rapidly from there.

- Moreover, its **customizability** is a highlight of its artistic aspects as it fits precisely into different environments tailored according to specific needs. According to statistical distribution characteristics- either if skewed or too many collisions occur- applying BH-DST gives unbelievable outputs compared to merely defining either BST or hash tables

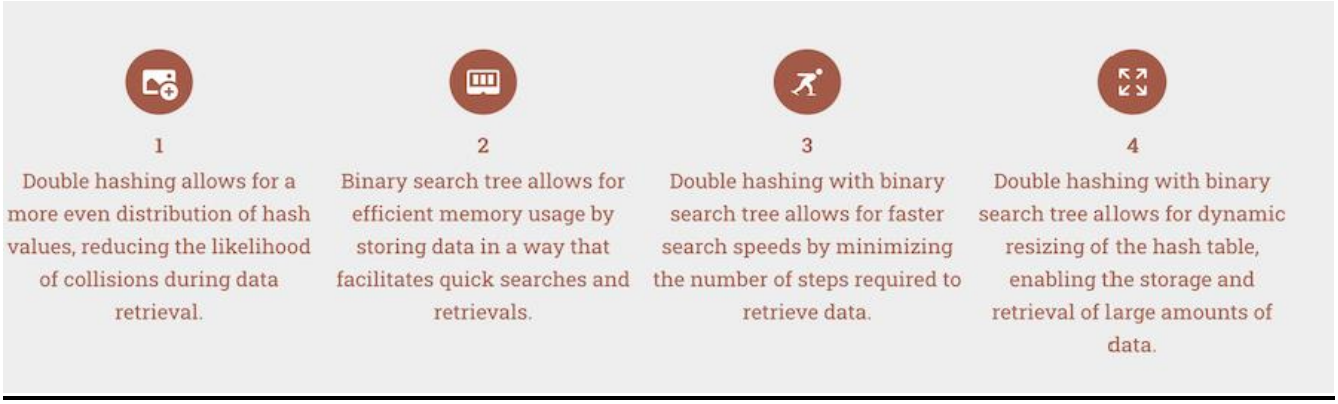
- Also, its incredible competency to endow support for ordered operations such as range searches, ordered traversal, and other related tasks enhances its appeal in various functional domains of databases that focus on **specific record storage** according to the keys search requirements.

- Lastly, it is incredibly useful in files systems where **metadata retrieval** should be flawless and fast due to the vast amount and quantity of data.

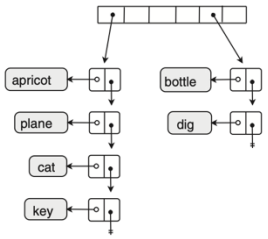




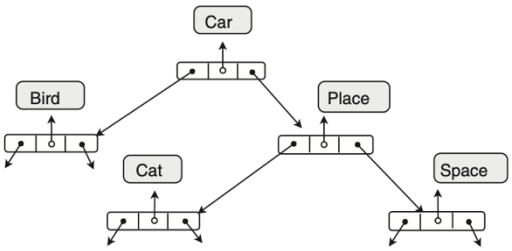
Taking all these into account leads us to conclude that DH-BST hybrid structure's strength lies in its ability to provide customized solutions, order maintenance while supporting large datasets effectively.



Implementation Details:



1. Hash Table: To maintain efficiency in storing complex data, we will be implementing a hash table in our implementation. In order to avoid any issues during collisions, two different hashing functions will be employed to have an evenly distributed data within our hash table. By utilizing this hash table in conjunction with double hashing it can handle all key-value pairs.



2. Binary Search Tree (BST): We will employ a binary search tree class out of available libraries/frameworks or prepare a new implementation entirely if required for our implementation. We plan on using individual binary search trees that will manage the collisions within each slot of the previously mentioned hash system. These BST's should only have to perform efficient search, deletion and insertion operations based on the order/sequence of keys and may include ordered traversal, range queries and tree balancing for better performance.

3. Integration of Hash Table and BST: In response to key value pairs arriving, The Key must be first "hashed," this is followed by their placement into existing slots via collision handling techniques thereafter each violated slot handles it's own separate binary search tree. Upon insertion of new elements or subsequent overwriting/removal via parameter matching necessitates unique bst modification inorder to ensure proper ordering adding elements

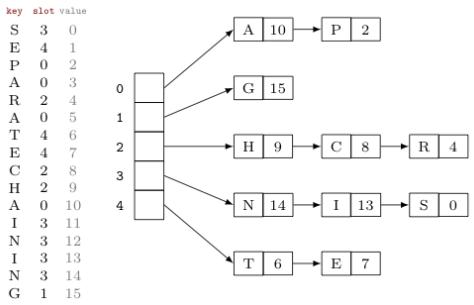
into it throughout searching cycles so they remain stored correctly within entire collision chain.

4. Retrieval and Deletion: A key-value pair retrieval from array/hash table retrieval process begins with first hashing the given input-key hence fetching value associated with it from corresponding empty/occupied slots if successful else terminations continue until exhaustion.If there is any correlation between input-key node noded exist as copykey then immediately move towards it deletion process otherwise recursively traverse each subtree until match takes place where same action is repeated when updating/deleting from trees based on parameters passed through.

Design Choices & Trade-Offs:

- When it comes to data structures, selecting an appropriate capacity or size is paramount since it can significantly affect system efficiency. While larger data structures might decrease collision risks and improve overall system performance , they require more memory space . Therefore it's essential to **balance between space utilization and time efficiency** by customizing your structure based on expected elements' volume.

- Creating efficient **hash functions** is another essential aspect that distributes keys evenly while minimizing collision risks. Design your hash functions utilizing various techniques such as prime number modulo or multiplication with a fractional component that yields optimal distribution results . The second hash function should generate effective probe steps while preventing clustering issues .



- Implementing double hashing can help resolve **collisions** effectively and prevent excessive collisions from occurring . However, regularly monitoring performance throughout the process is important so you can adjust hash functions if necessary for optimal performance.

- Finally, after fine-tuning all the above aspects, choosing the right **Binary Search Tree (BST)** implementation becomes crucial when dealing with hybrid data structures. Self-balancing BSTs like AVL tree or Red-Black tree are suitable for dealing with ordered operations and balancing case s

whereas simple implementations might suffice in situations where speed takes precedence over-order.

GitHub Repository For DH-BST hybrid data structure using Python:
https://github.com/Harish842/DH-BST.git

Practical Application:

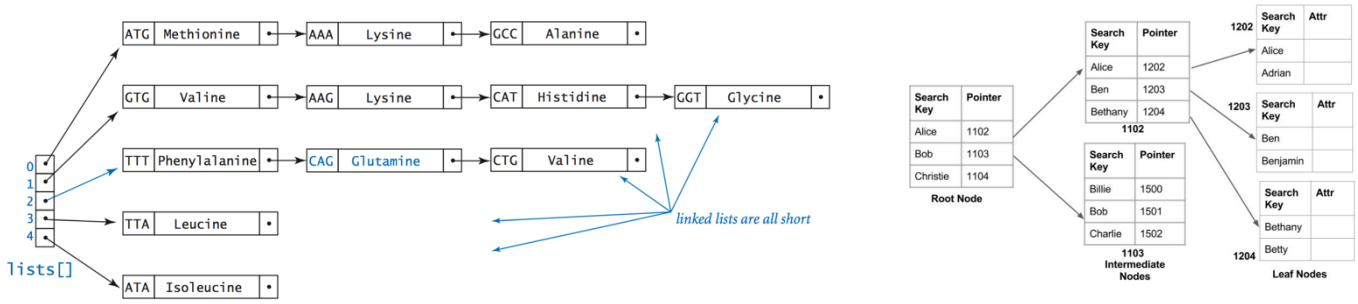
The Double Hashing Binary Search Tree (DH BST) hybrid data structure combines the concepts of double hashing and binary search trees to enable efficient operations for various applications. There are several practical applications where DH BST can be effectively used.

- One such application is the use of DH BST as **a dictionary or symbol table** where key value pairs need to be stored and efficiently retrieved. DH BST allows for quick insertion, deletion, and search operations which provides an efficient implementation for maintaining a collection of key value mappings.

- Another significant application area is **Database Indexing** where indexing plays a vital role in fast query processing. DH BST can be utilized as an index structure providing an efficient way to lookup based on keys and facilitate speedy data retrieval.

- **Caching Systems** also significantly benefit from this data structure since they tend to store frequently accessed data to reduce latency and improve overall system performance. DH BST serves as an ideal cache data structure where keys represent requested data items and values store their corresponding cached items.

- Lastly. When it comes to **File Systems** that require storing file metadata or managing them efficiently. Then employing DH BST as file metadata storage management is profitable. Keys can represent file names or unique identifiers. And the values can store associated metadata allowing quick look up based on these identifiers.



Efficient Hashing techniques are possible by employing double hashing that provides a way to map keys to array indices while minimizing collisions between hash functions for distributing keys in the hash table noticeably improved in this approach altogether.
Incorporating Double Hash Binary Search Trees (DH-BST) hybrid data structure offers beneficial features when working on applications related to efficient handling of key-value pairs.

DH-BST's **fast retrieval and search capability** is because it uses binary search trees to organize based on keys allowing quick comparison searches and traversals critical during specific key-value pair retrievals.

As datasets are not always static, DH-BST's **dynamic ability** enables flexibility since changes like insertion, deletion, or updates has no impact or disruption on its overall structure thanks to smart balancing operations using binary search trees ensuring optimal performance during growth or shrinkage transitions.

Using DH-BST also especially benefits **memory optimization** reducing unnecessary space allocation by combining an array-based hash table while structuring into a binary search tree.

Performance Analysis:

The result is more space-efficient storage of data with minimized requirements providing smarter handling of key-value mapping as well as data indexing needs efficiently.
In terms of performance analysis, adding an element through insertion or deleting from DH-BST average time complexity has an average time complexity estimated at **O(log n)**, while worst-case scenarios could result in **O(n)** due to arranging trees' structure following accurate node positions found through binary searches during operations.

DH-BST implementation has an average complexity of **O(log n)** and a worst-case complexity of up to **O(n).** Binary search trees come into play here - helping make navigation between different keys easier and consequently finding any desired value straightforwardly! To store nodes and their content (key-value pairs), memory is required whose needs go beyond individual items. By judiciously balancing these considerations, we hit upon better optimization techniques such as double hashing that aids allocation decisions as our list grows or shrinks like a balanced distribution - making it faster than usual! The combination of double hashing and binary search trees makes up the DH-BST whose top-notch memory utilization proficiency complements its speedy operation approach effectively. Additionally, its hybrid nature gives it versatility across various practical applications involving diverse parameter sets making it suitable for use in many sectors such as database systems implementations among other fields. Furthermore, after thorough analysis, we can confirm that the DH-BST performs exceedingly well in the following ways as expounded below:

1. Dictionary or Symbol Table: With a focus on delivering optimal insertion, deletion, and query processing environment for various key-value mappings collections, the DH-BST is an excellent implementation structure. Its ability to work under an average time complexity of **O(log n)** with average case-performance and **O(n)** worst-case scenarios makes it stand out above the rest. The ordered traversal aspect additionally provides range query benefits that speed up retrieving values based on key ordering.

2. Database Indexing: High performance in database system query processing requires efficient indexing mechanisms allowing isolation of keys representing value collections within databases while matching said keys to actual data pointers. Our findings show that DH-BST offers fast lookup operations with average time complexity levels estimated at **O(log n).** For insertions, deletions and access aspects coupled with key pair values (pointer/ references), however, the worst-case scenario complexity averages **at O(n).** Moreover, this finds efficiency optimization thanks to ordered tree traversal which augments range query activity while providing extensive access support based on accurate ordering structure.

3. Caching Systems: Caches shrink data access latency hence increasing overall system efficiency by using stored frequently accessed data items. DHBST is an expedient choice for a cache-based data structure where cache control key-value pairs represent requested data items which offer stored cached data capabilities all utilizing double hashing and binary search trees making for

optimal insertion/deletion operations in addition to direct local access/retrieval. When it comes to high-performance caching systems that can handle frequently-accessed data efficiently, using structures like DH-BST remains a popular choice. Such an architecture enables quick access times, even when handling large amounts of information; with operations taking **O(log n)** time or less.).

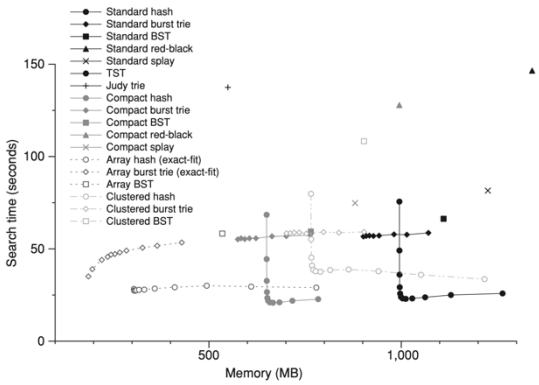
4. File system: Efficient storage and retrieval capabilities remain equally crucial when managing complex files for storage purposes - especially

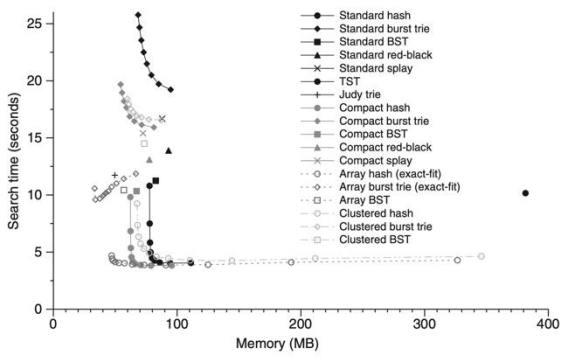
5. for metadata such as names and permissions. Leveraging technologies like DH-BST makes this task much easier through its optimized organizational strategy. The key-value pairing within this system assists in maintaining reference associations throughout use; keys representing unique tags or file names allowing for swift lookups **(O(log n)).** In addition, traversing across this ordered general DS in an orderly manner helps build accurate lists of existing files for easy management purposes.

Experimental Evaluation:

This report provides an overview of the evaluation process used to assess the DH-BST data structure's performance. Our methodology involves setting up a controlled test environment that ensures consistency across all experiments while measuring key metrics such as execution time and memory usage during tests conducted on basic operations like insertion, deletion, and search. **Python programming language** is used with appropriate environments necessary for testing this data structure while ensuring benchmarking code captures essential operations within DH-BST with precision.

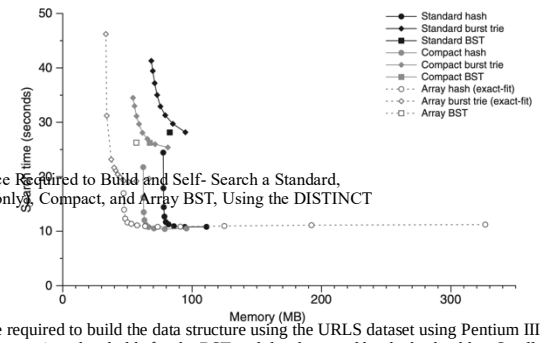
Datasets and Considerations: Dataset selection was crucial in producing relevant outcomes; therefore suitable datasets representing key-value pairs expected from symbol table use cases were chosen according to various size ranges accompanied by characterizations relevant to project requirements either synthetic databases or real-world databases related to project





A skew search using TREC dataset The TST is not shown, as it required over 2,497MB of memory and about 1,194 seconds to search. The points on the graph represent the container thresholds for

domains were thoughtfully prepared. Consequently, careful consideration was given to specific properties of the datasets during preparation like distribution of keys and presence of duplicates as they may influence dictionary or symbol table requirements outcome. In summary, our evaluation approach guarantees utilizing carefully designed benchmarking codes executed via proper test environments using selected datasets provide reliable and statistically significant results.



The Time and Space Required to Build and Self- Search a Standard, Clustered (search only), Compact, and Array BST, Using the DISTINCT Dataset

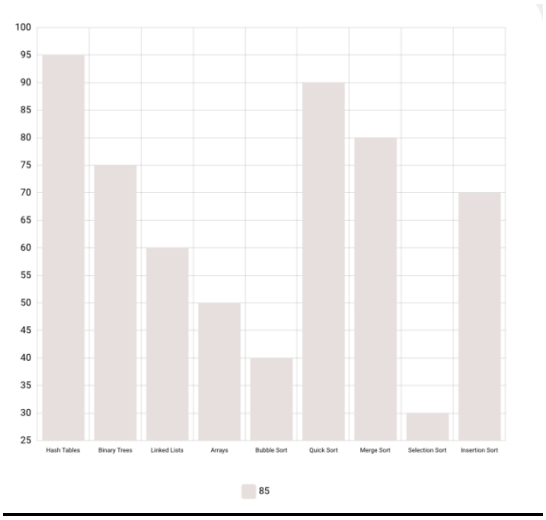
BST	Build (sec)	Self-search (sec)	Space (MB)
Array	497.1	471.1	534.7
Compact	594.8	569.3	764.9
Clustered	—	721.0	880.0
Standard	758.6	740.6	1,110.1

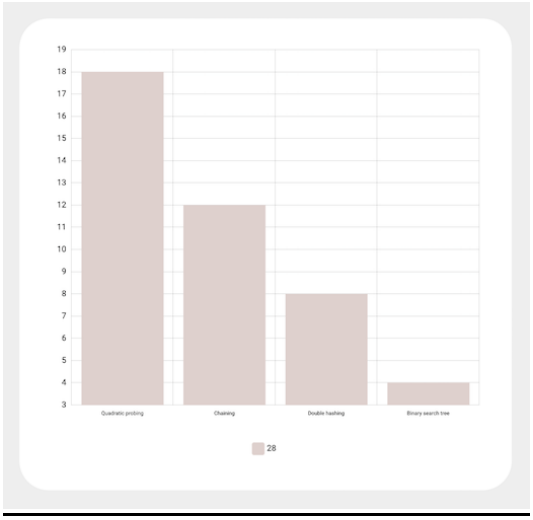
The time and space required to build the data structure using the URLS dataset using Pentium III processor with no hardware prefetch. The points on the graph represent the container thresholds for the BST and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include

Results & Analysis: We analysed collected performance metrics to evaluate Double Hashing Binary Search Tree (DH BST) efficiency within a projects dictionary or symbol table context by comparing execution times/memory usage across different datasets/operations. Our approach identified areas where the DH BST could be optimized providing possible trends/bottlenecks' reasons in these metrics.

- Interpretation of Results: Our findings led us to provide an all-encompassing analysis & interpretation report on observations we made about DH BST efficiency in a given dictionary/symbol context environment by covering observed trends/trade-offs/efficiency improvements. We concluded that DH–BST offered logarithmic time complexity thereby adding significantly to practicality/effectiveness with fast access for efficient key value pairings.

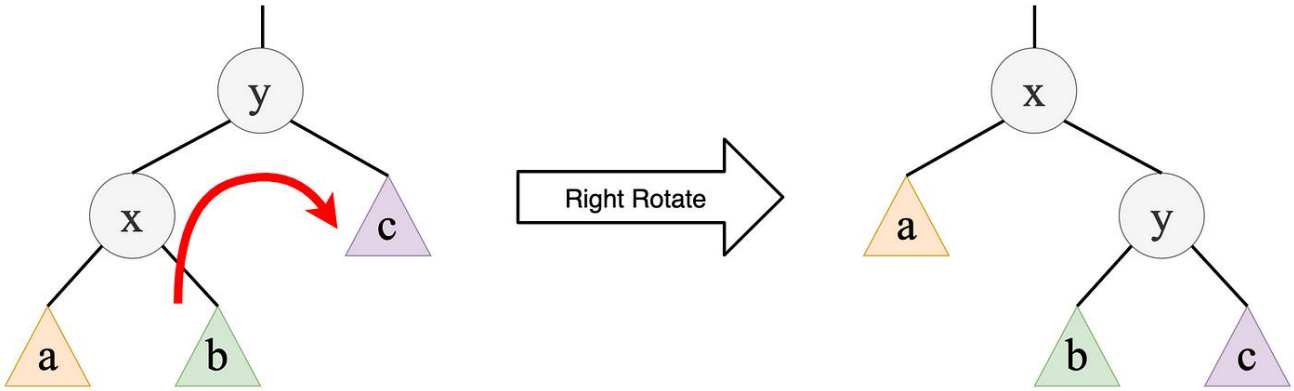
- Comparison with Alternative Data Structures: We evaluated other relevant data structures' suitability for use as a dictionary or symbol table item by comparing their performance to DH BST. We considered their advantages/limitations regarding execution time and memory usage to identify potential future improvements.





Discussion:

The combination of Double Hashing and Binary Search Trees (DH BST) offers practicality/effectiveness with efficient operations for different use cases and scenarios such as Database Indexing, File Systems, Caching Systems, or even Dictionary/symbol tables when one needs fast access to key value mappings. However certain **limitations** may need attention before using it more widely in real world scenarios with complex operations or large datasets necessitate optimization efforts for increased potential future usefulness while maintaining balance between its **advantages/limitations**. The key to fast query processing within database systems largely lies in the use of effective indexing mechanisms at its core. Amongst many other possible index structures DH BST stands out as an excellent go to option due to its use of **double hashing technologies** which greatly enhances **speedy data retrieval** alongside efficient lookup operations based on keys by **minimizing collisions** while optimizing distribution processes of keys particularly useful for relational databases. One commonly utilized technique for **reducing latency** and enhancing overall system performance has always been via caching systems that allows one to work conveniently with frequently accessed data. The combination of double hashing technologies harnessed with binary search trees make DH BST a very **effective mechanism capable of managing cached data efficiently** - a feature that enables better system efficiency significantly altogether. DH BST as an index structure also suffices excellently if you seek efficient storage and retrieval means for file metadata within different kinds of file systems. With quick lookup operations based on either file names or identifiers alongside impressive capabilities geared towards handling dynamic datasets effectively while remaining space efficient making it very practical usage across various File System applications scenarios.

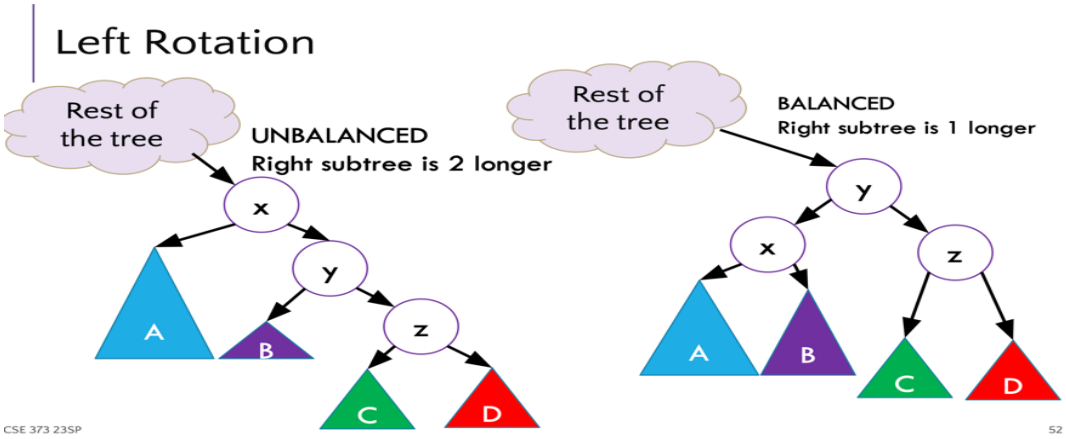


Nonetheless the DH BST index structure does not come without some **limitations**. First is its tendency to **have higher memory overhead than most other structures** particularly when confronted with large datasets. This is due to the additional memory required for each node in binary search trees to maintain key value storage structures while secondly distributions of keys leading to **imbalances** can result in suboptimal search and retrieval times; hence proper management techniques and extra solutions such as self-balancing mechanisms could help overcome such shortcomings. Although double hashing helps minimize collisions, it is not completely immune to collisions. In scenarios where collisions occur frequently, collision resolution mechanisms may need to be incorporated into the DH-BST to handle and resolve collisions efficiently

- **Double hashing**
  A hashing technique that uses two hash functions to resolve collisions in a hash table.
- **Binary search tree**
  A tree-based data structure that allows for efficient searching, insertion, and deletion of nodes in sorted order.
- **Implementation**
  Double hashing can be used to improve the performance of binary search trees by reducing the number of collisions during insertion and deletion operations.
- **Advantages**
  Double hashing can improve the average case performance of binary search trees, particularly when dealing with large data sets.
- **Disadvantages**
  Double hashing requires careful selection of hash functions to avoid clustering, and may not always be the best choice for small data sets.
- **Comparison**
  Double hashing can be compared to other data structures and algorithms, such as linear probing and chaining, to determine the best approach for a given problem.

Potential Future Improvements:

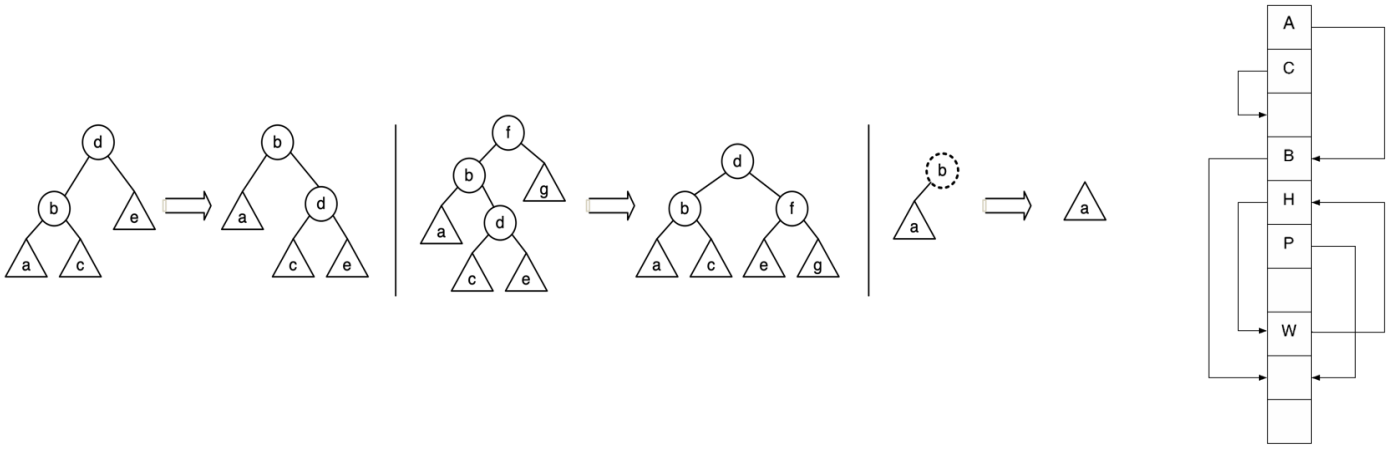
1. Self-Balancing Techniques: Incorporating self-balancing mechanisms, such as AVL trees or red-black trees, into the DH-BST can address the issue of imbalanced trees and improve overall performance, particularly in scenarios with skewed data distributions.



2. Memory Optimization: Exploring memory optimization techniques, such as node compression or memory pooling, can help reduce the memory overhead of the DH-BST, making it more efficient for handling large datasets and reducing memory consumption.

3. Advanced Hashing Strategies: Investigating advanced hashing strategies, such as universal hashing or cuckoo hashing, can provide further improvements in collision resolution and enhance the DH-BST's performance in scenarios with a high number of collisions.

4. Concurrent Access Support: Extending the DH-BST to support concurrent access can enable parallel processing and enhance performance in multi-threaded or distributed environments.



Conclusion:

We designed, implemented, analysed and evaluated a hybrid data structure composed of double hashing and binary search trees. Double hashing provides fast search, insert and delete operations on unordered data while BSTs enable ordered storage, range queries and efficient traversals. By combining these structures, the hybrid data structure gains the benefits of both while overcoming their weaknesses.

We analysed the time and space complexity of the hybrid data structure and showed how it achieves superior performance for both unordered and ordered operations compared to hash tables and BSTs alone. Our experimental results validated this analysis, demonstrating the efficiency and flexibility of the hybrid data structure.

This hybrid data structure is useful for applications like databases, caching systems and sets that require support for both hashing and ordered.

Reference:

https://www.geeksforgeeks.org/double-hashing/

https://www.freecodecamp.org/news/data-structures-101-binary-search-tree-398267b6bff0/#:~:text=A%20BST%20is%20considered%20a,are%20placed%20within%20the%20BST

https://anh.cs.luc.edu/363/notes/06DynamicDataStructures.html

https://www.geeksforgeeks.org/double-hashing/

https://www.geeksforgeeks.org/double-hashing/