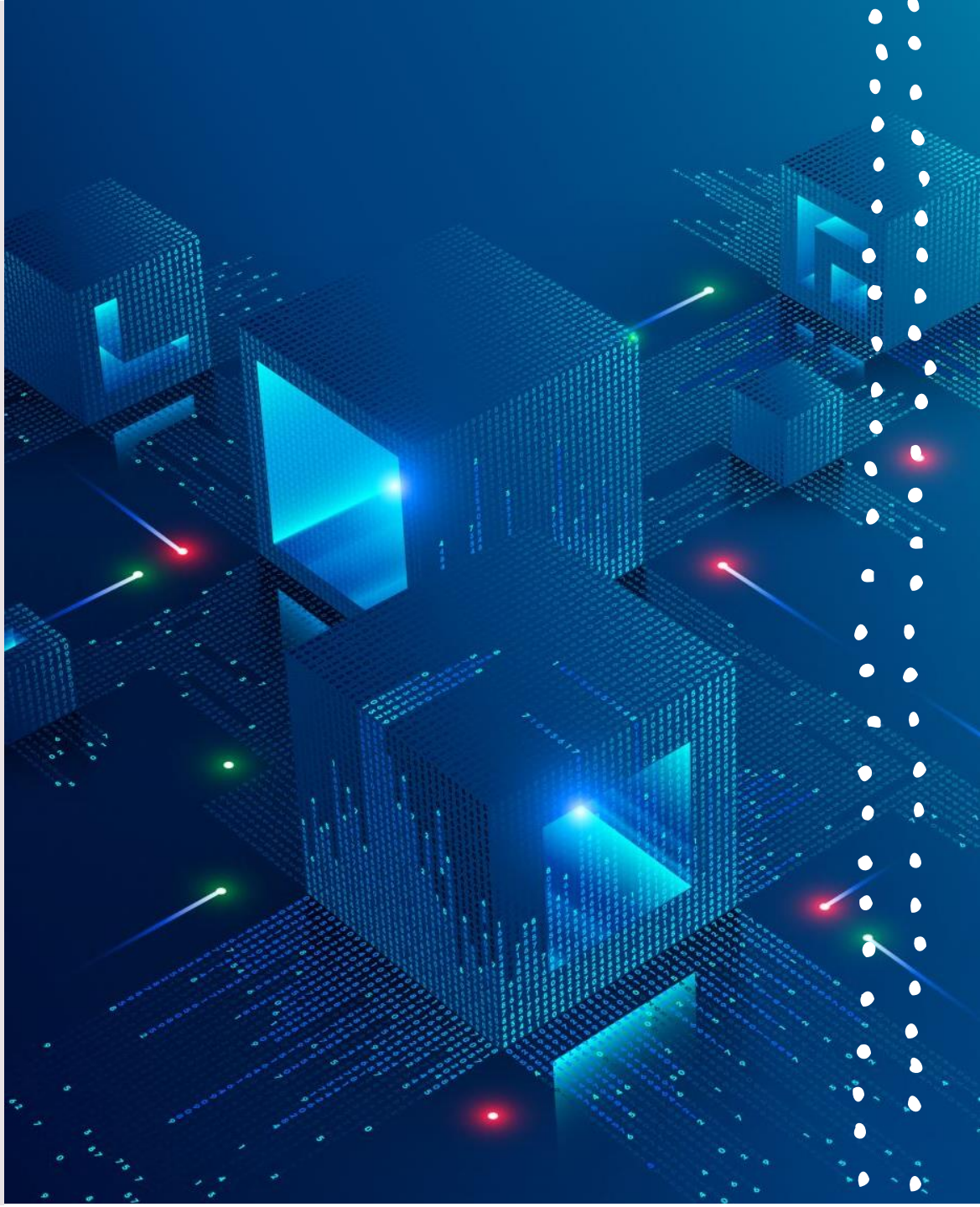
HYBRID DATASTRUCTURE

ALGOZEN

S No.	NAME	ROLL NO.
1	Adwaita Sathrukkan	CB.EN.U4CSE2403
2	Bapathi Archana	CB.EN.U4CSE2410
3	Golla Thanuja	CB.EN.U4CSE2419
4	Harish S	CB.EN.U4CSE2422
5	Sivakami V	CB.EN.U4CSE2456

What is hybrid data-structure?

- Hybrid data structures refer to data structures that combine the characteristics or features of multiple different data structures.
- They are designed to provide the advantages and capabilities of multiple data structures in a single structure, allowing for efficient operations and improved performance for specific use cases.



Double Hashing with Binary Search Tree (BST)

EXPLORING A COLLISION RESOLUTION TECHNIQUE

Introduction to Double Hashing

01

Double hashing is a collision resolution technique commonly used in hash tables.

02

It involves using two hash functions instead of one to determine the position of an element in the hash table.

03

h1(k) must hash the same type of keys as h2(k).where h1 andh2 are hash functions.

04

The size of the hash table must preferably be prime number.

WORKING

Assume a hash table size as 10

Let the keys to be inserted be

3,2,9,6,11,7,12

Let h1(k)=2k+3 and h2(k)=3k+1

i traversing from 0 to 9

V is the collision free index, i.e, first free index from (location+V*i)%m

V=h2(k)%m

key	location	V
3	(2*3+3)%10=9	-
2	(2*2+3)%10=7	-
9	(2*9+3)%10=1	-
6	(2*6+3)%10=5	-

WORKING

For key 11 the hash index is 5 which is already filled with key 6.

Now we must use the second hash function and find the V value

Which is h2(k)=3k+1,which means:

V(11)=[(3*11)+1]=4

Loc(11)=(5+4*0)%10=5
Loc(11)=(5+4*1)%10=9
Loc(11)=(5+4*2)%10=3

0	
1	9
2	
3	11
4	
5	6
6	
7	2
8	
9	3

key	location	V
3	(2*3+3)%10=9	-
2	(2*2+3)%10=7	-
9	(2*9+3)%10=1	-
6	(2*6+3)%10=5	-
11	(2*11+3)%10=5	

WORKING

For key 7 the hash index is 7 which is already filled with key 2.

Now we must use the second hash function and find the V value

Which is h2(k)=3k+1,which means:

V(7)=(3*7+1)%10=2

Loc(7)=(7+2*0)%10=7

Loc(7)=(7+2*1)%10=9

Loc(7)=(7+2*2)%10=1

Loc(7)=(7+2*3)%10=3

Loc(7)=(7+2*4)%10=5

Loc(7)=(7+2*5)%10=7

Loc(7)=(7+2*6)%10=9

Loc(7)=(7+2*7)%10=1

Loc(7)=(7+2*8)%10=3

Loc(7)=(7+2*9)%10=5

0	
1	9
2	
3	11
4	
5	6
6	
7	2
8	
9	3

key	location	CFI
3	(2*3+3)%10=9	-
2	(2*2+3)%10=7	-
9	(2*9+3)%10=1	-
6	(2*6+3)%10=5	-
11	(2*11+3)%10=5	3
7	(2*7+3)%10=7	

WORKING

For key 12 the hash index is 7 which is already filled with key 2.

Now we must use the second hash function and find the V value

Which is h2(k)=3k+1,which means:

V(12)=(3*12+1)%10=7

Loc(12)=(7+7*0)%10=7
Loc(12)=(7+7*1)%10=4

0	
1	9
2	
3	11
4	12
5	6
6	
7	2
8	
9	3

key	location	CFI
3	(2*3+3)%10=9	-
2	(2*2+3)%10=7	-
9	(2*9+3)%10=1	-
6	(2*6+3)%10=5	-
11	(2*11+3)%10=5	3
7	(2*7+3)%10=7	-
12	(2*12+3)%10=7	4

- # Time Complexity :

  - Each lookup in the hash-set costs O(1) constant time.
  - In the worst case we might not find such a compliment and iterate through the entire array, this would cost us O(n) linear time.
  - Rest operations like subtraction are mere constant-time operations.
  - Hence total time complexity = O(n) time.

- # Space Complexity:

  - We need to maintain an extra hash-set of size upto n elements which costs us extra O(n) space.
  - Hence total space complexity = O(n) time.

Introduction to Binary Search Tree(BST)

01

A binary search tree (BST) is a tree data structure that maintains sorted data.

02

The nodes of a BST are arranged in such a way that the key of each node is greater than or equal to the keys of all its left children and less than or equal to the keys of all its right children.

03

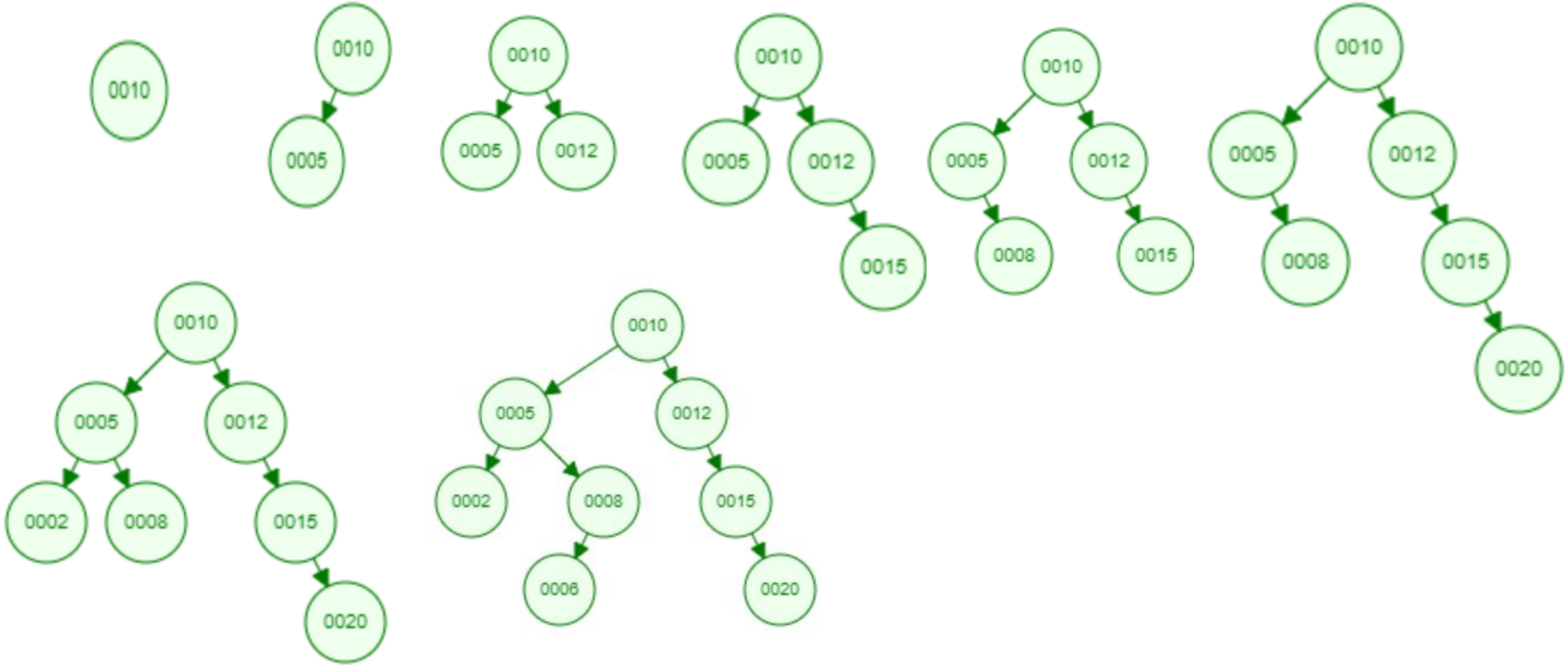
The topmost node of the tree is called the root node, and it is the starting point for accessing any element in the tree. The nodes beneath the root are arranged in a hierarchical manner, forming the branches and leaves of the tree.

04

BSTs are a very efficient data structure for searching, insertion, and deletion operations. The time complexity of these operations is O(log n), where n is the number of nodes in the tree.However, if the tree becomes unbalanced, the worst-case time complexity can degrade to O(n), where n is the number of nodes in the tree.

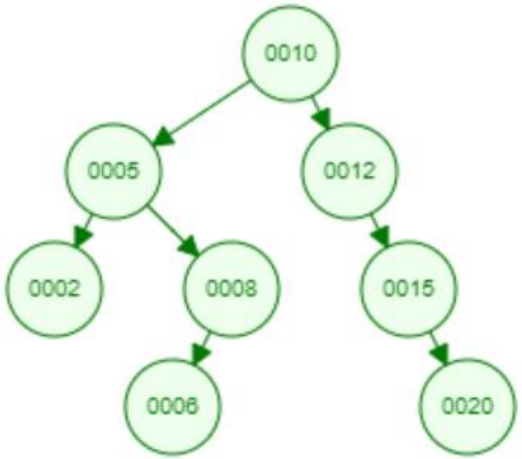
WORKING

Let us consider values 10,5,12,15,8,20,2,6 for creating a Binary Search Tree

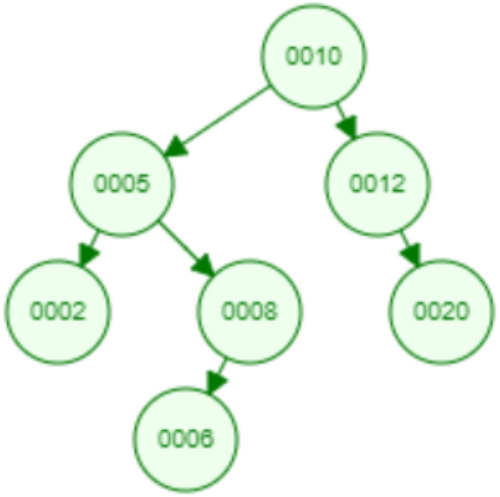


WORKING

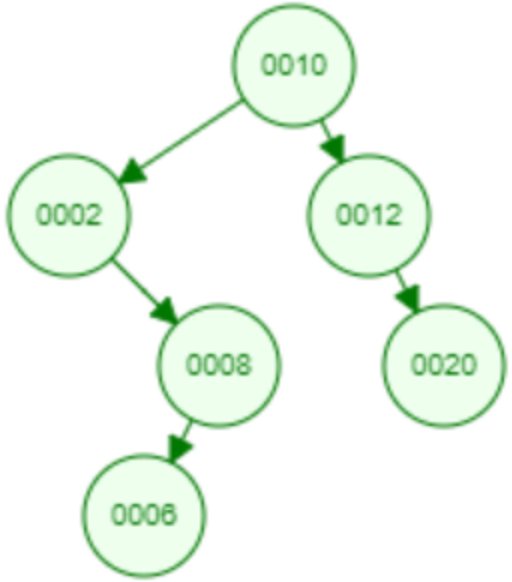
Deletion in Binary Search Tree



- Delete 15



- Delete 5

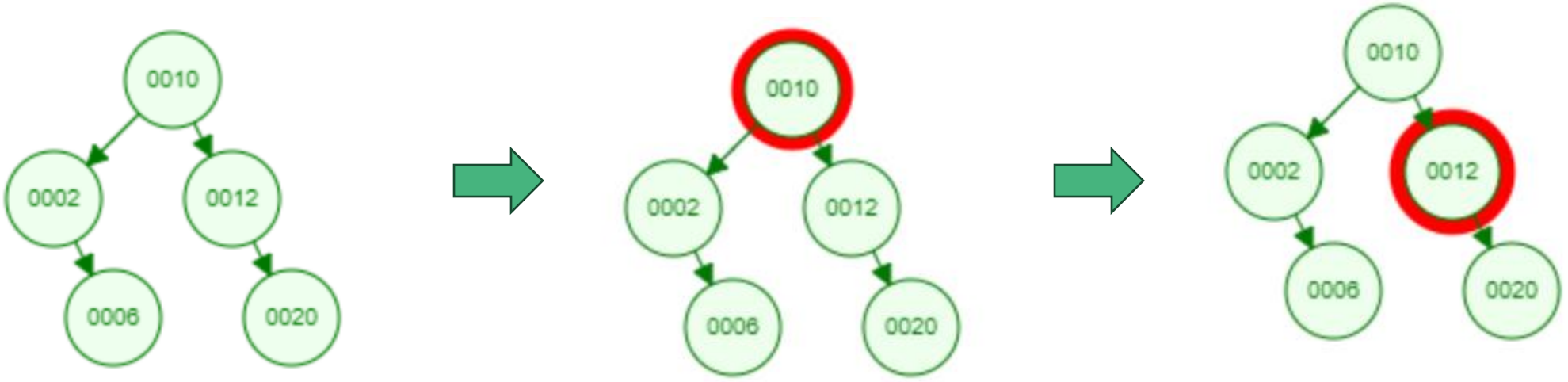


- Delete 8

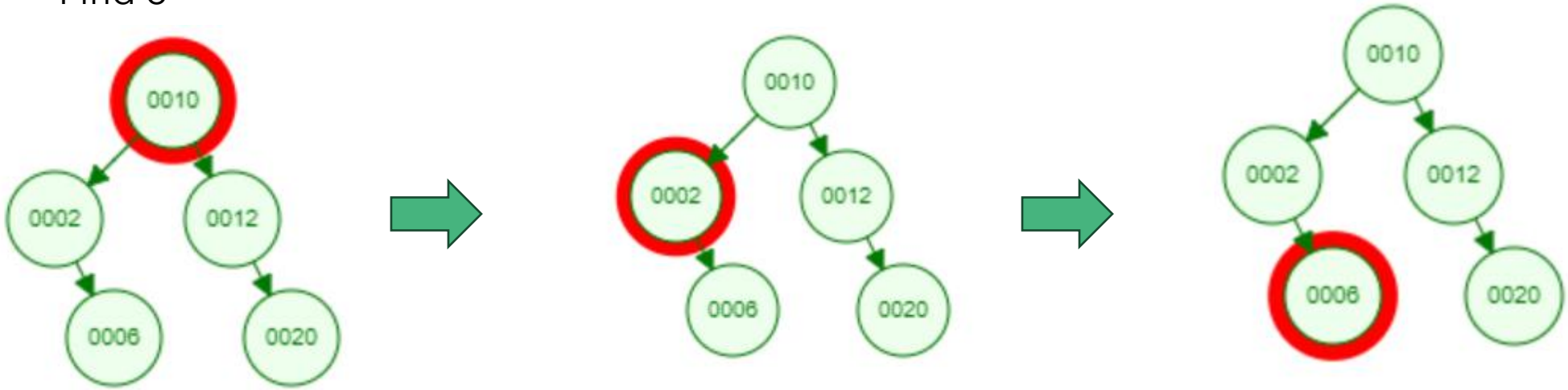
WORKING

Searching for an element Binary Search Tree

- Find 12



- Find 6



Double hashing with Binary Search tree

01

Double hashing allows for a more even distribution of hash values, reducing the likelihood of collisions during data retrieval.

02

Binary search tree allows for efficient memory usage by storing data in a way that facilitates quick searches and retrievals.

03

Double hashing with binary search tree allows for faster search speeds by minimizing the number of steps required to retrieve data.

04

Double hashing with binary search tree allows for dynamic resizing of the hash table, enabling the storage and retrieval of large amounts of data.

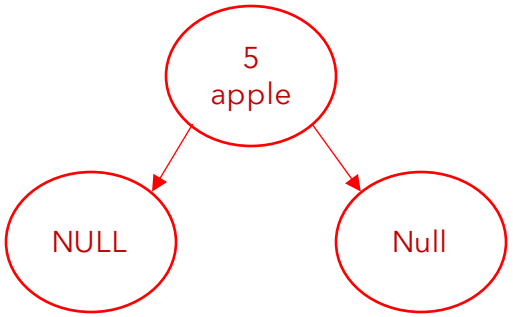
Hash Table

Slot 1:
Slot 2:
Slot 3:
Slot 4:
Slot 5:

- The diagram represents a hash table that uses the Double Hashing technique to handle collisions. Each slot in the hash table contains a binary search tree (BST) to store key-value pairs.
- In the example diagram, there are five slots in the hash table, labelled Slot 1 to Slot 5. Each slot represents a position in the hash table where key-value pairs can be stored.
- The use of Double Hashing helps resolve collisions by providing an alternative hash function that can be used if the primary hash function results in a collision. This allows for spreading the key-value pairs across different slots in the hash table, reducing the chances of collisions and improving overall performance.

WORKING ctd..

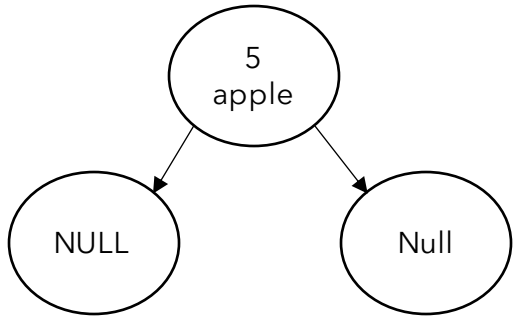
Hash Table

Slot 1: 
Slot 2:
Slot 3:
Slot 4:
Slot 5:

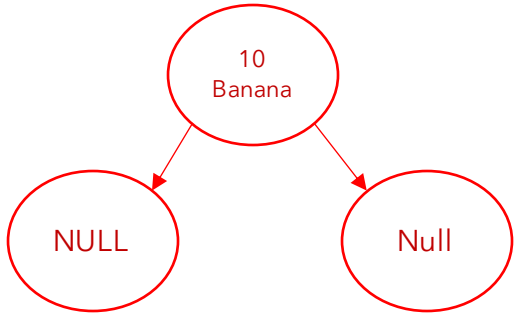
- To insert the key-value pair ("apple", 5):
  - The key "apple" is hashed to determine the slot (Slot 1).
  - Since Slot 1 is empty, the pair ("apple", 5) is directly placed in Slot 1.

Hash Table

Slot 1:



Slot 2:



Slot 3:

Slot 4:

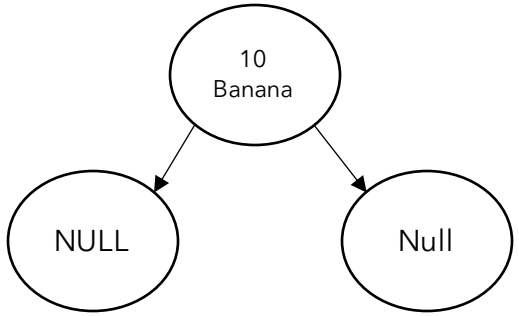
Slot 5:

To insert the key-value pair ("banana", 10):

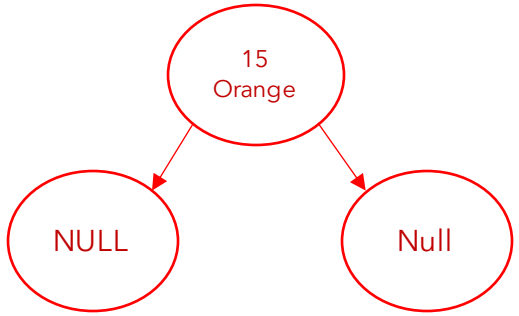
The key "banana" is hashed to determine the slot (Slot 2).

Since Slot 2 is empty, the pair ("banana", 10) is directly placed in Slot 2.

Slot 2:



Slot 3:



Slot 4:

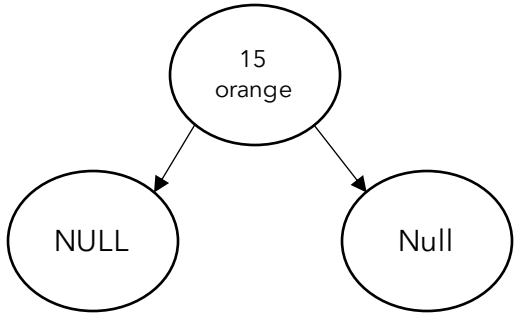
Slot 5:

To insert the key-value pair ("orange", 15):

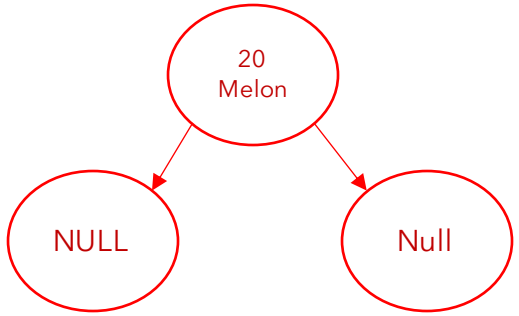
The key "orange" is hashed to determine the slot (Slot 3).

Since Slot 3 is empty, the pair ("orange", 15) is directly placed in Slot 3.

Slot 3:



Slot 4:



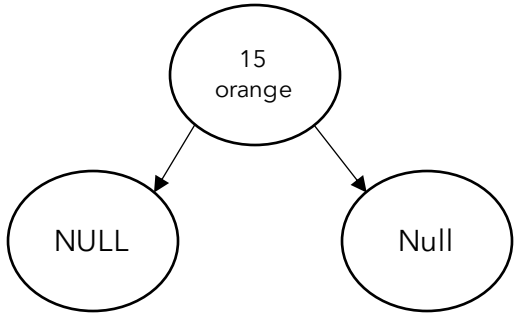
Slot 5:

To insert the key-value pair ("melon", 20):

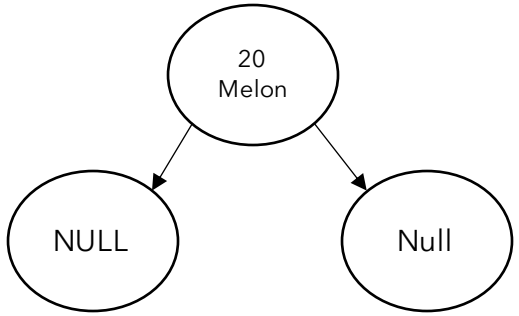
The key "melon" is hashed to determine the slot (Slot 4).

Since Slot 4 is empty, the pair ("melon", 20) is directly placed in Slot 4.

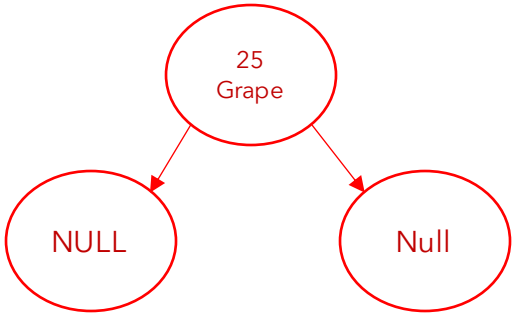
Slot 3:



Slot 4:



Slot 5:

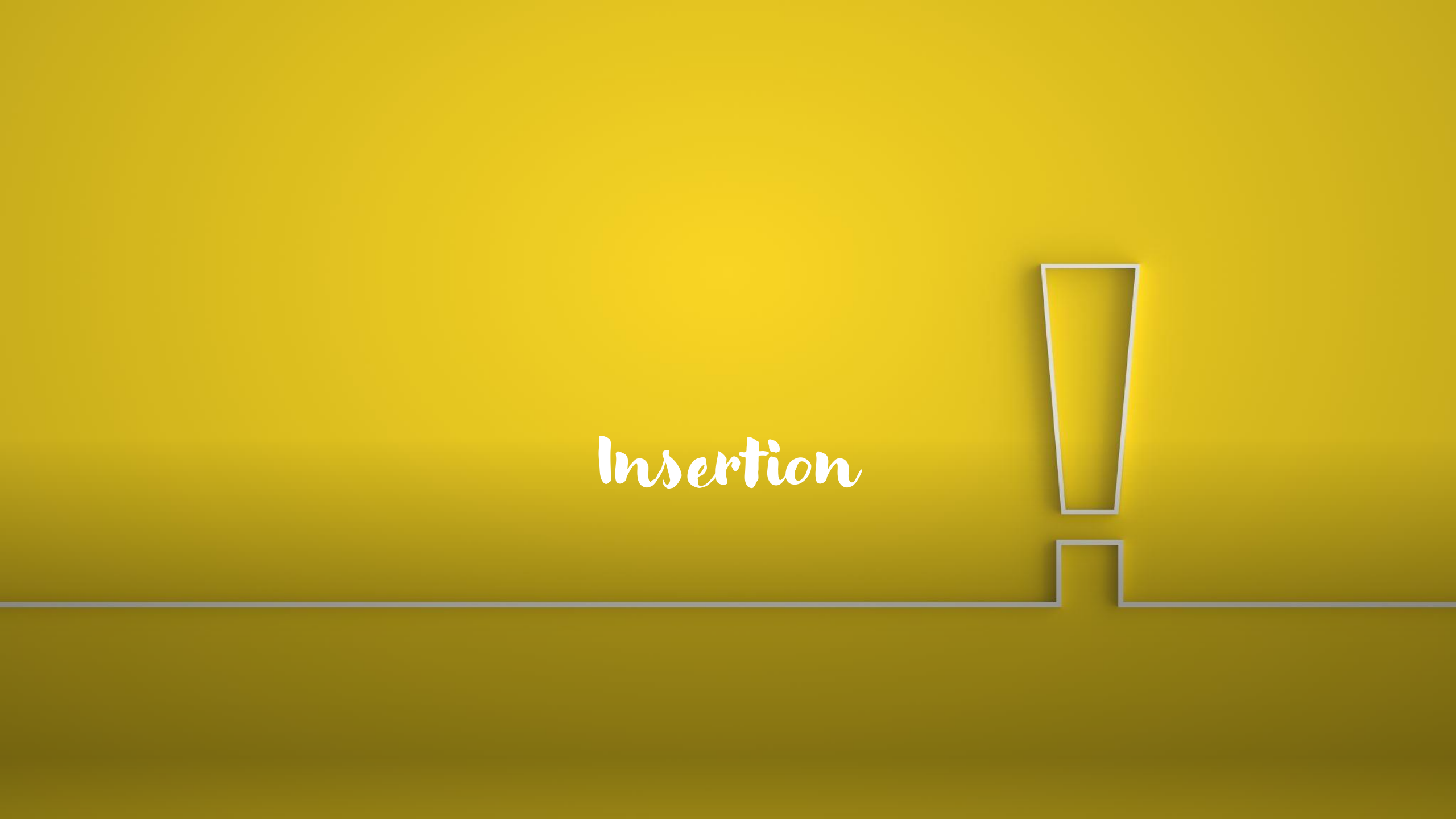


To insert the key-value pair ("grape", 25):

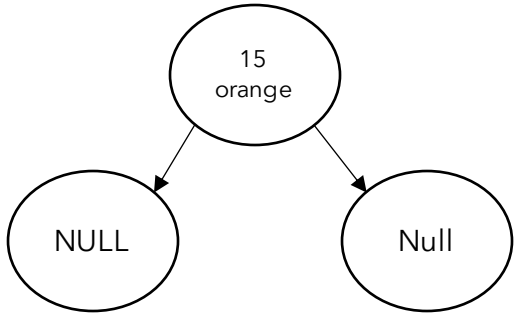
The key "grape" is hashed to determine the slot (Slot 5).

Since Slot 5 is empty, the pair ("grape", 25) is directly placed in Slot 5.

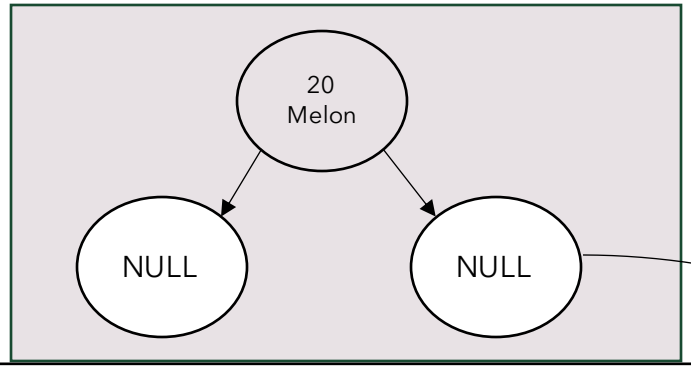
Insertion



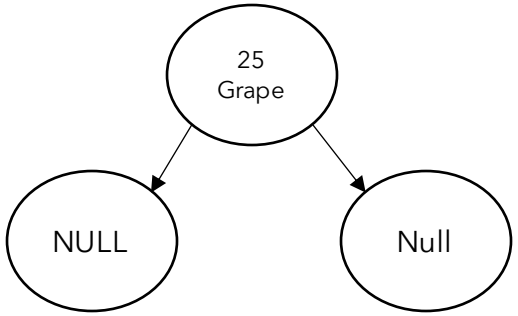
Slot 3:



Slot 4:



Slot 5:

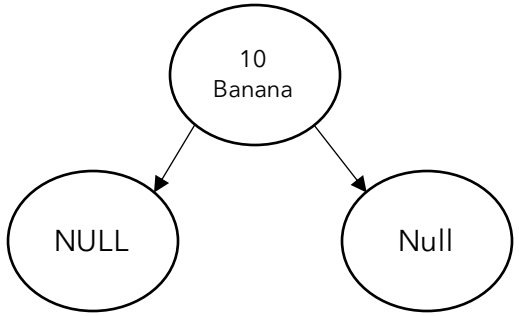


Insertion:

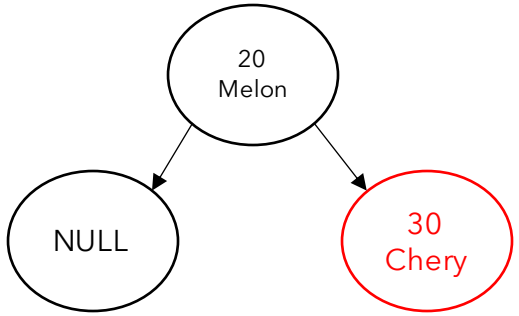
- Insert the key-value pair ("cherry", 30).
- Hash the key "cherry" to determine the slot (Slot 4).
- Since Slot 4 is already occupied, we use Double Hashing and update it.
- Insert the key-value pair ("cherry", 30) into the BST of Slot 4.

Insert Here

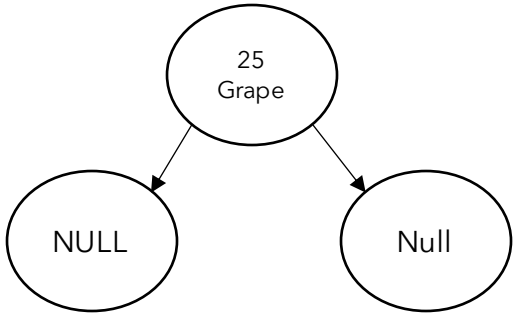
Slot 3:



Slot 4:

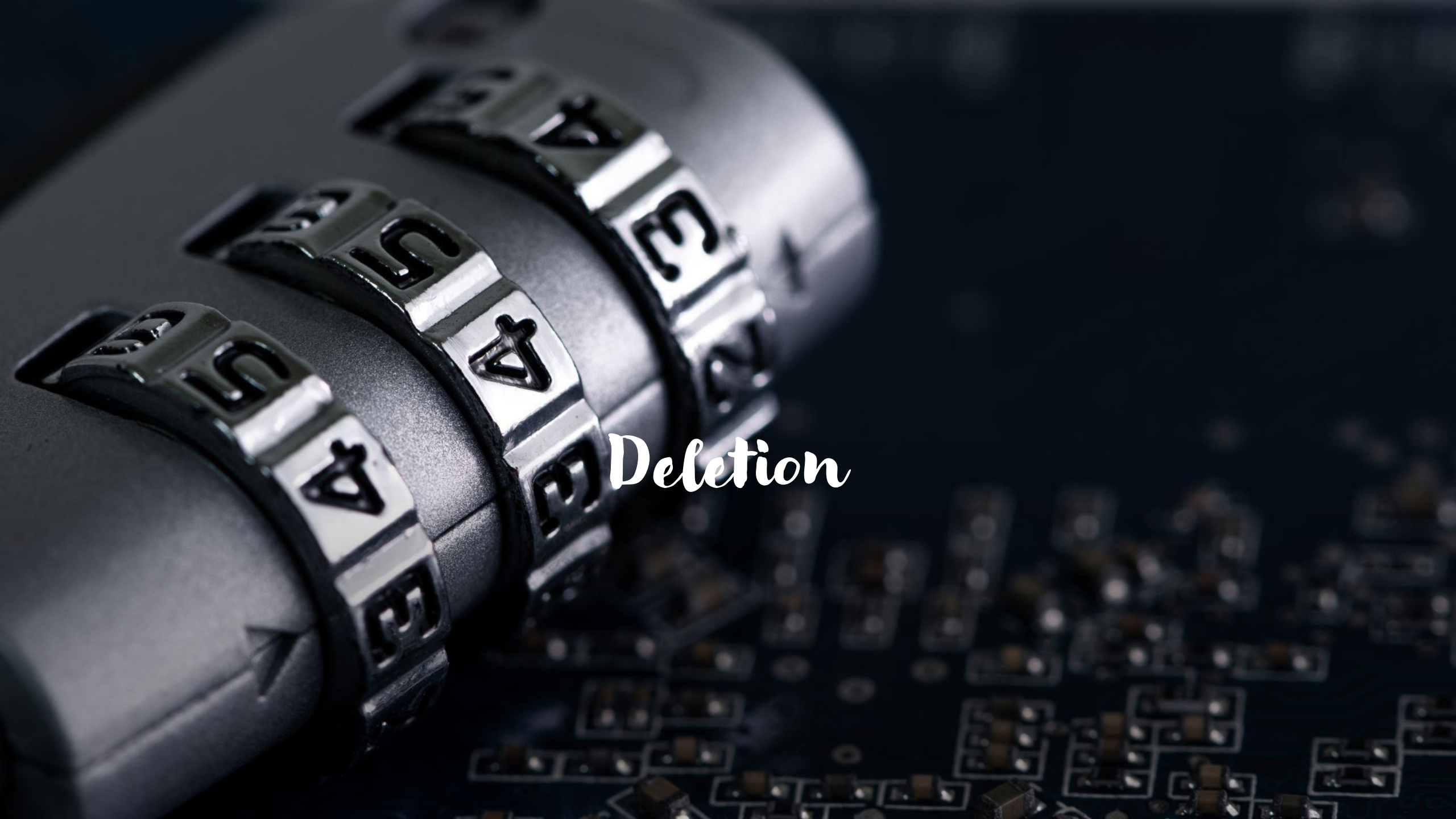


Slot 5:

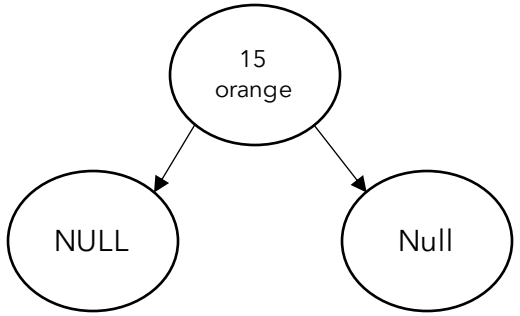


Insertion:

- Insert the key-value pair ("cherry", 30).
- Hash the key "cherry" to determine the slot (Slot 4).
- Since Slot 4 is already occupied, we use Double Hashing and update it.
- Insert the key-value pair ("cherry", 30) into the BST of Slot 4.

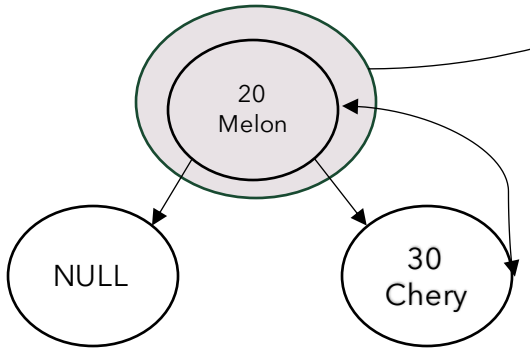

Deletion

Slot 3:



Remove it

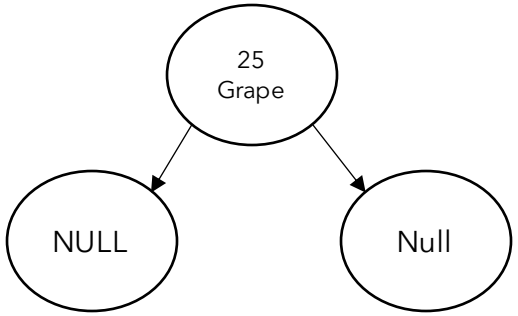
Slot 4:



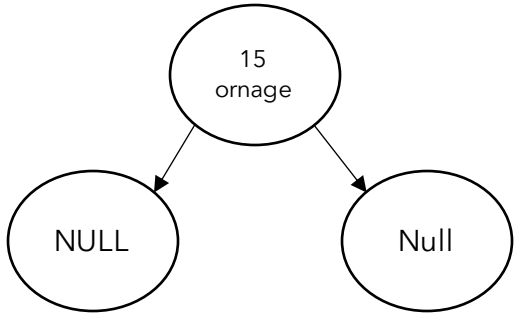
Deletion:

- Delete the key-value pair (”melon", 20).
- Hash the key ”Melon" to find the slot (Slot 4).
- Search for the node with the key ”melon" in the BST of Slot 4.
- Remove the node while maintaining the BST's ordering.

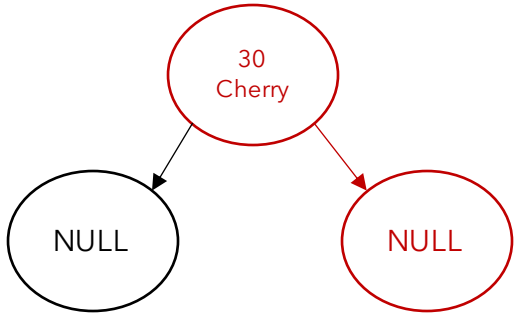
Slot 5:



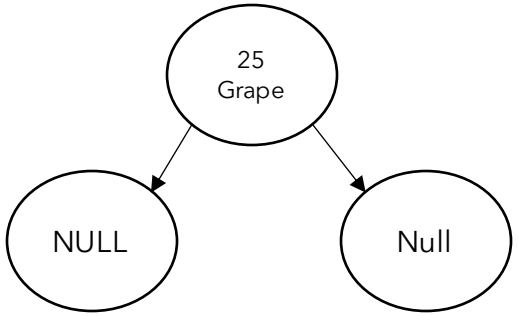
Slot 3:



Slot 4:



Slot 5:

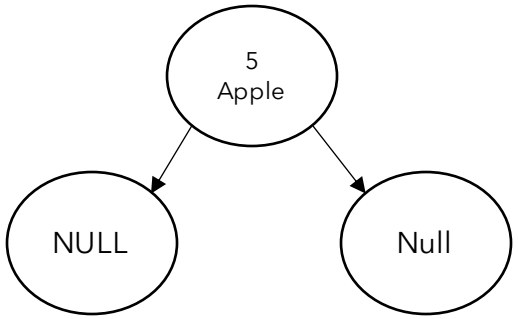


Deletion:

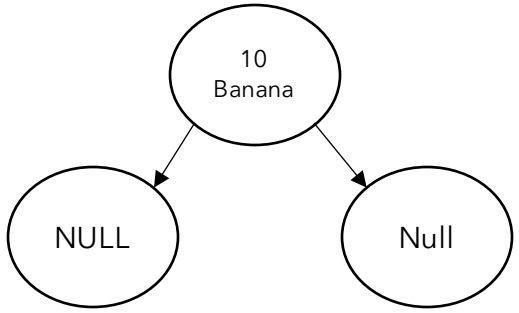
- Delete the key-value pair (”melon", 20).
- Hash the key ”Melon" to find the slot (Slot 4).
- Search for the node with the key ”melon" in the BST of Slot 4.
- Remove the node while maintaining the BST's ordering.


Searching

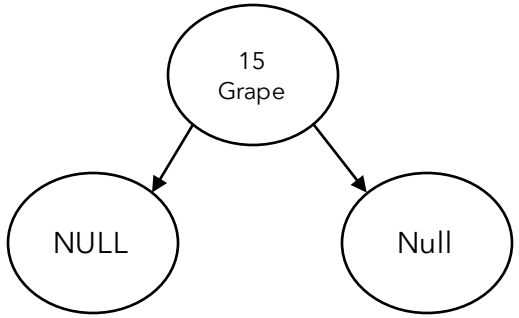
Slot 1:



Slot 2:



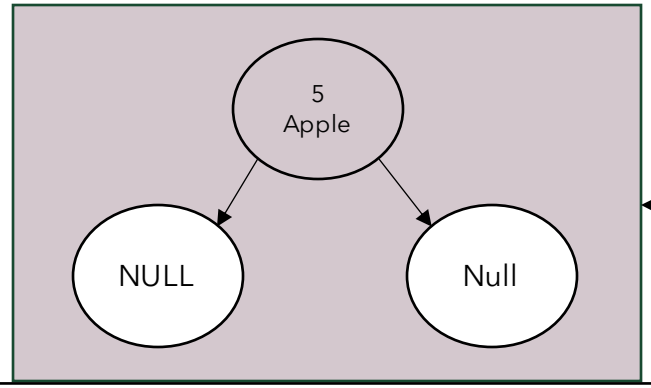
Slot 3:



Searching:

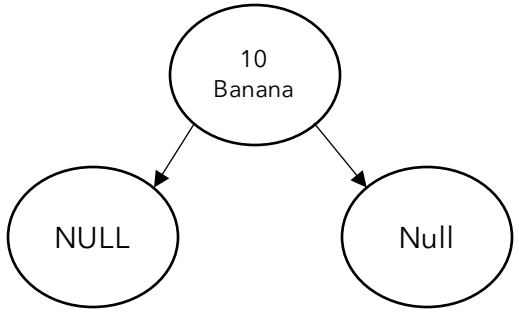
- Search for the key ”Grape".
- Hash the key "melon" to determine the slot (Slot 5).
- Search for the node with the key "melon" in the BST of Slot 5.

Slot 1:

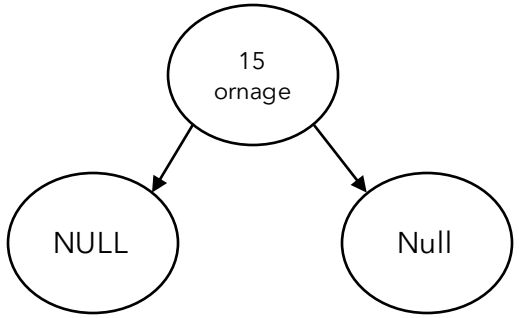


NO, Move next

Slot 2:



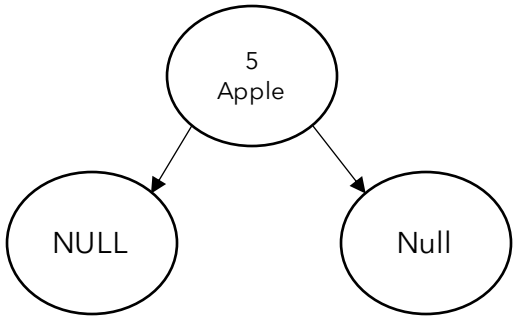
Slot 3:



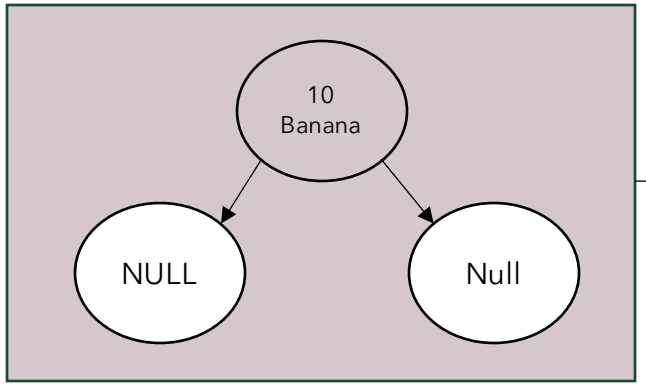
Searching:

- Search for the key ”Grape".
- Hash the key "melon" to determine the slot (Slot 5).
- Search for the node with the key "melon" in the BST of Slot 5.

Slot 1:

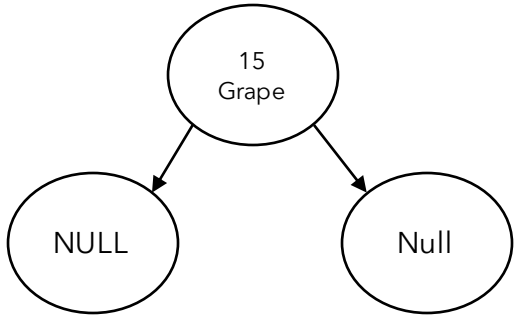


Slot 2:



NO, Move next

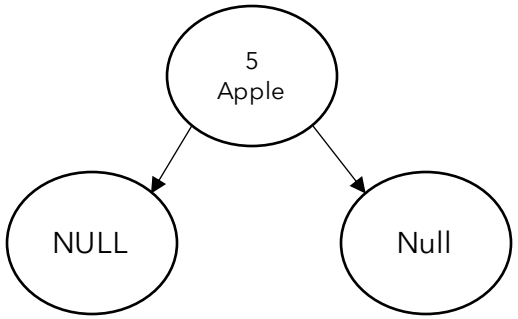
Slot 3:



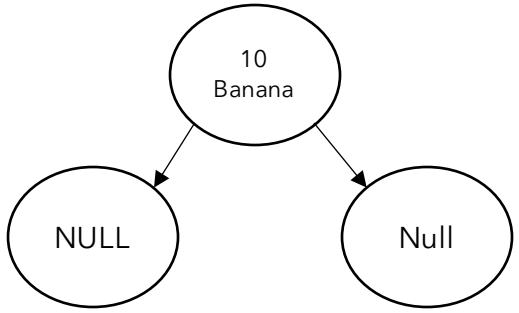
Searching:

- Search for the key ”Grape".
- Hash the key ”Grape" to determine the slot (Slot 5).
- Search for the node with the key "melon" in the BST of Slot 5.

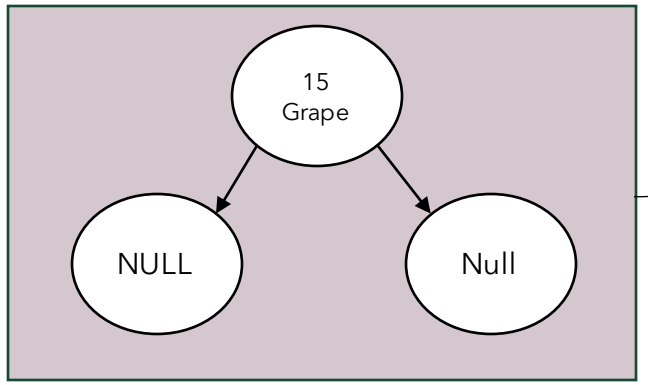
Slot 1:



Slot 2:



Slot 3:

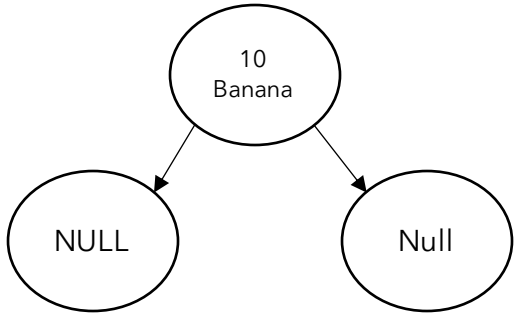


NO, Move next

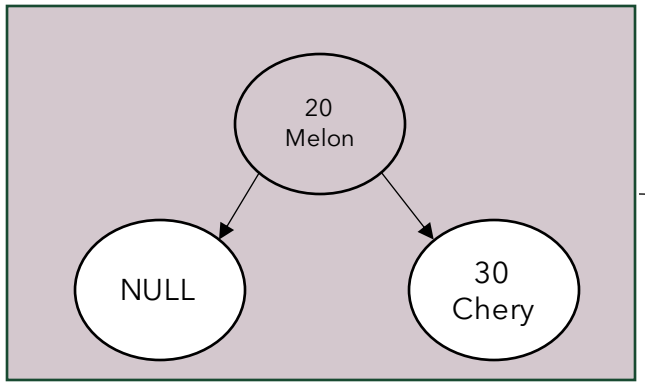
Searching:

- Search for the key ”Grape".
- Hash the key "melon" to determine the slot (Slot 5).
- Search for the node with the key "melon" in the BST of Slot 5.

Slot 3:

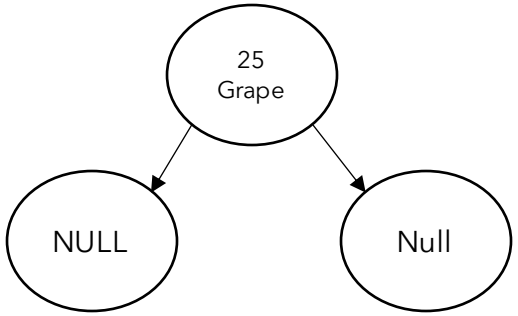


Slot 4:



NO, Move next

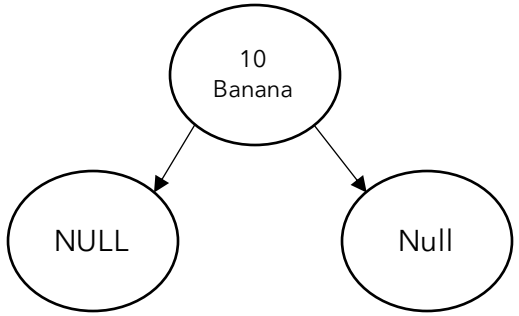
Slot 5:



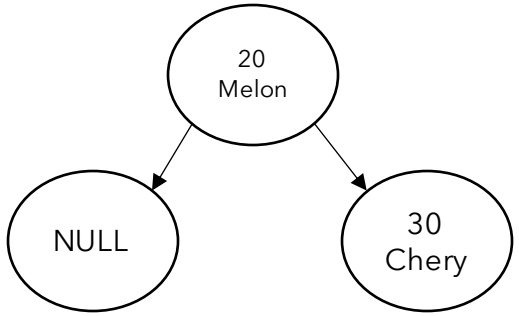
Searching:

- Search for the key ”Grape".
- Hash the key "melon" to determine the slot (Slot 5).
- Search for the node with the key "melon" in the BST of Slot 5.

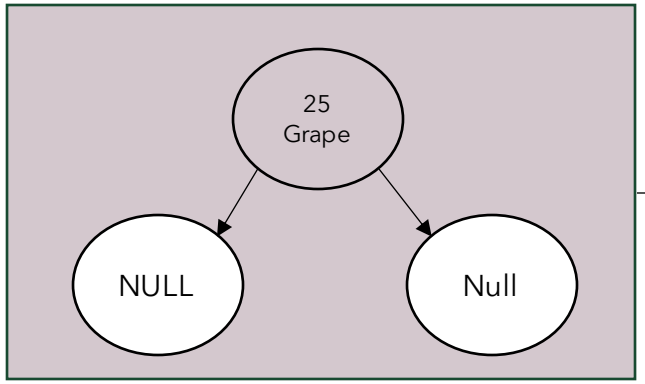
Slot 3:



Slot 4:



Slot 5:



Searching:

- Search for the key ”Grape".
- Hash the key "melon" to determine the slot (Slot 5).
- Search for the node with the key "melon" in the BST of Slot 5.

Searching Operation done

- # Time Complexity :
  - Insertion: O(1) average case, O(n) worst case (when there are many collisions and the BST becomes unbalanced)
  - Deletion: O(1) average case, O(n) worst case (when there are many collisions and the BST becomes unbalanced)
  - Searching: O(1) average case, O(n) worst case (when there are many collisions and the BST becomes unbalanced)
- # Space Complexity:
  - The space complexity of the hash table itself is O(n), where n is the number of key-value pairs inserted.
  - The space complexity of the binary search trees within each slot depends on the number of elements in each BST and the structure of the BSTs.

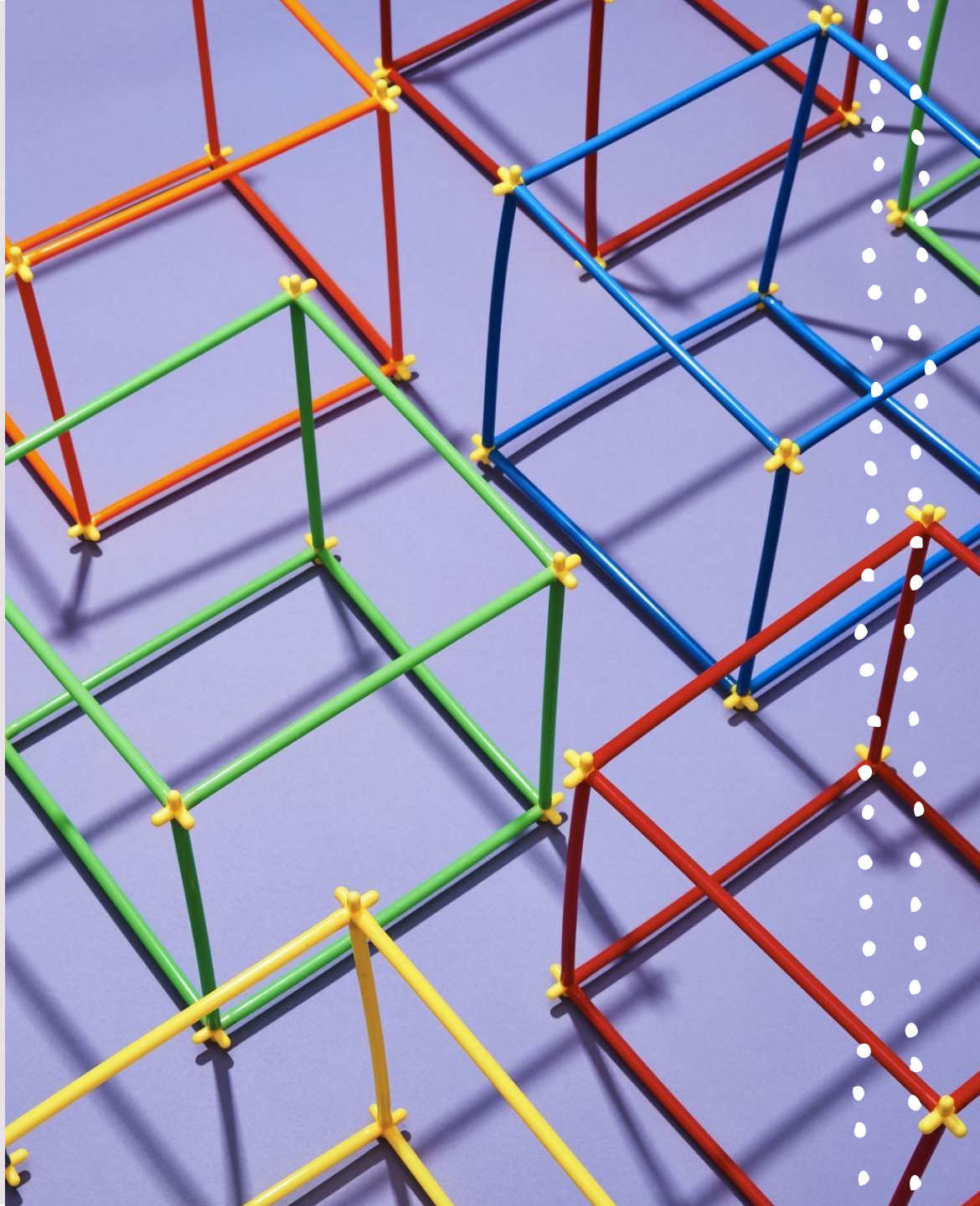
Practical Application

- 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.

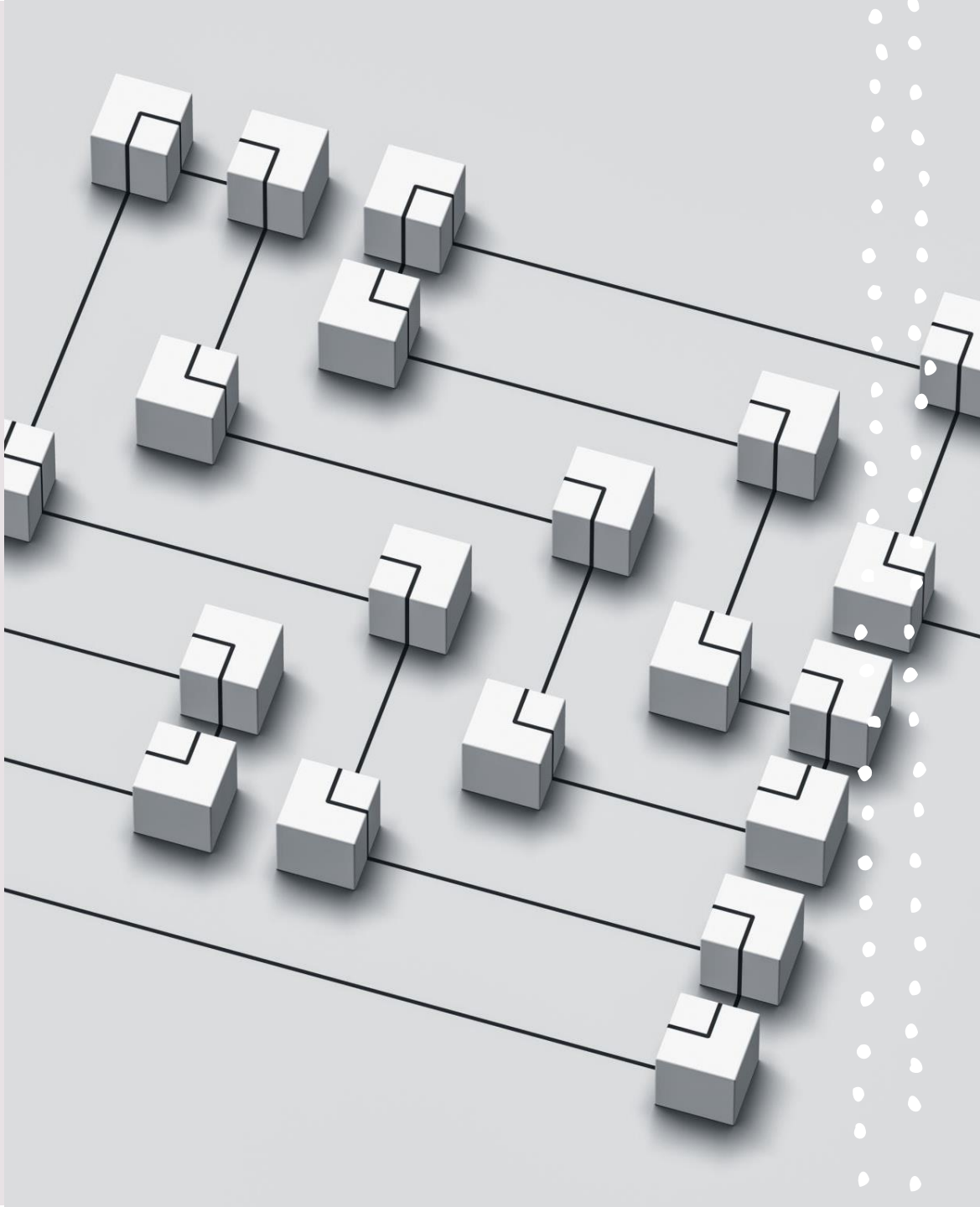


Performance Analysis

- 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.
- 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.
- Caching Systems: Caches shrink data access latency hence increasing overall system efficiency by using stored frequently accessed data items.
- File system: Efficient storage and retrieval capabilities remain equally crucial when managing complex files for storage purposes - especially

Experimental Evaluation

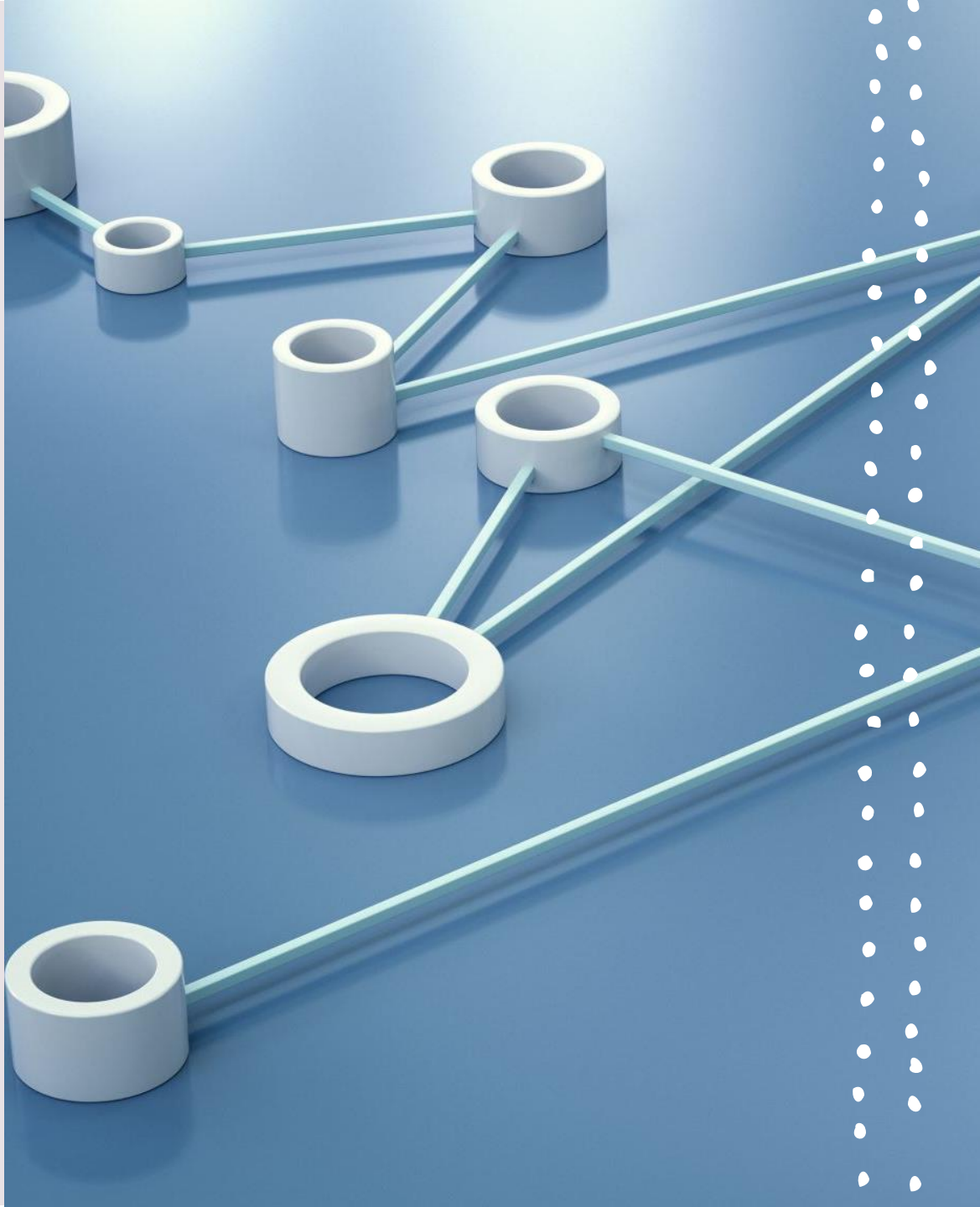
- **Python programming language** is used with appropriate environments necessary for testing this data structure while ensuring benchmarking code captures essential operations within DH-BST with precision.
- Datasets and Considerations: Dataset selection was crucial in producing relevant outcomes; therefore 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 domains were thoughtfully prepared.



- 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.
- 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.
- 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.

Conclusion

- 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




Thankyou..