

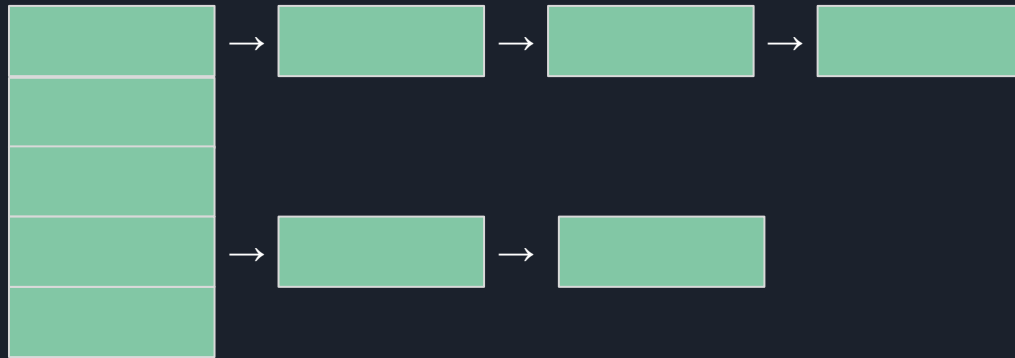
A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front parallelogram is blue and the back one is a light green color. Both are oriented diagonally from the top-left towards the bottom-right.

# Efficient Hashing Using Huffman Coding



# Hashing using Chaining

**Definition:** A HashMap is a collection of key-value pairs. It uses an array and a Linked List internally for storing Key and Value.





# Hashing using Chaining

## Insertion in a HashMap:

- A Hash function is used to generate a hash value for the input key.
- Example Hash Function for an input string:

```
def hashFunction(key):  
    hash = 7  
    for k in key:  
        hash = hash*1000000007 + k  
    return hash
```

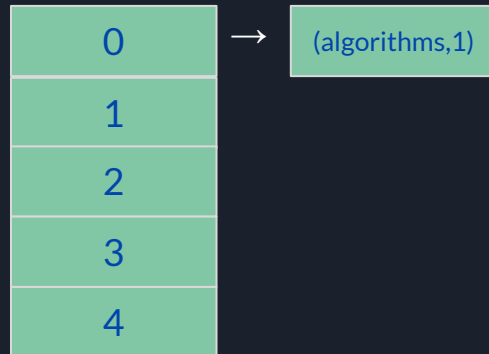
- The index is the hash value generated modulo the size of the array.
- The key-value pair is appended to the linked list at the particular index



# Hashing using Chaining

## Insertion in a HashMap Example:

- Insert (algorithms,1): Hash(algorithms) -> 50, Index(50) -> 0

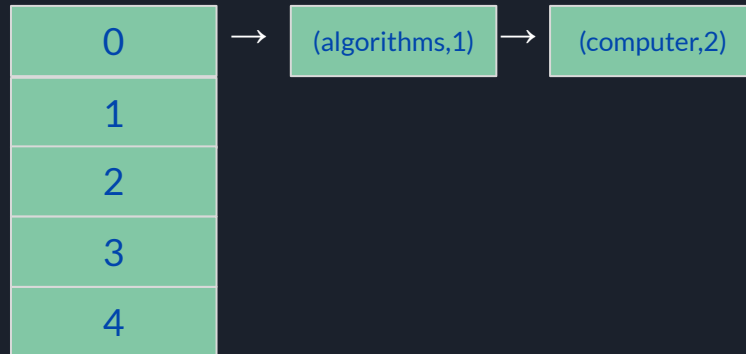




# Hashing using Chaining

## Insertion in a HashMap Example:

- Insert (computer,2): Hash(computer) -> 90, Index(90) -> 0

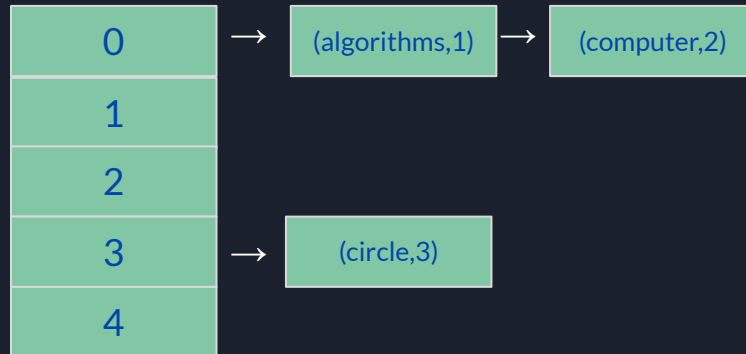




# Hashing using Chaining

## Insertion in a HashMap Example:

- Insert (circle,3): Hash(circle) -> 63, Index(63) -> 3





# Hashing using Chaining

- **Load Factor:** The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
- A load factor of 0.6 suggests that the HashMap can only be filled to 60% of its size i.e the  $\text{Capacity of the HashMap} = \text{Load Factor} \times \text{Size}$
- Once the HashMap is reaches its maximum capacity, the size of the table is increased to 1.5 times the previous size and the elements are rehashed and inserted into the new table.
- Hence the reduction of collisions!



# Hashing using Chaining

- **Time Complexity:** The insertion has an amortized time complexity of  $O(1)$  and the lookup time is  $O(1)$ .
- **Space Complexity:**  $O(1)$





# Huffman Coding

**Definition:** Huffman code is a type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code is Huffman coding.

**Input:** Alphabet  $A = \{a_1, a_2, a_3, \dots, a_n\}$  where  $n$  is the number of alphabets.

**Output:** Binary Code  $B = \{b_1, b_2, b_3, \dots, b_n\}$  where  $b_i$  is a binary code mapped to the alphabet  $a_i$ .

**Objective:**

- The objective is to create a Huffman Tree based, using the input characters and their frequency and assign them binary codes, and reduce the resultant bit size.
- Most commonly occurring characters are assigned the smaller bit codes and the least frequently characters are assigned the larger bit codes, to reduce the resultant bit size.

Example:

Character	Frequency	Binary Code
A	30	"00"
B	23	"10"
C	15	"11"
D	9	"100"



# Huffman Coding

Character	Frequency
A	5
B	25
C	10
D	50

## Algorithm to build a Huffman Tree:

- The leaves of the tree are the characters of the alphabet.
- Create a Max Heap(Priority Queue) and store all the characters of the input string along with their frequency.
- The top most element of the min heap is the least occurring character.



# Huffman Coding

Character	Frequency
A	5
B	25
C	10
D	50

Algorithm to build a Huffman Tree:

- Remove Two Nodes of Min Frequency from the Min Heap



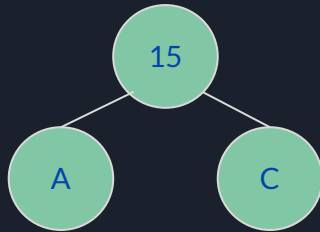


# Huffman Coding

Character	Frequency
A	5
B	25
C	10
D	50

Algorithm to build a Huffman Tree:

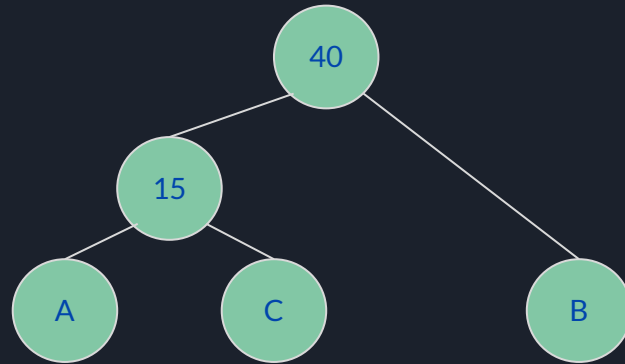
- Create a new internal node with frequency equal to the sum of it's two children and add it to the max Heap.
- Repeat the above steps until only one node remains.





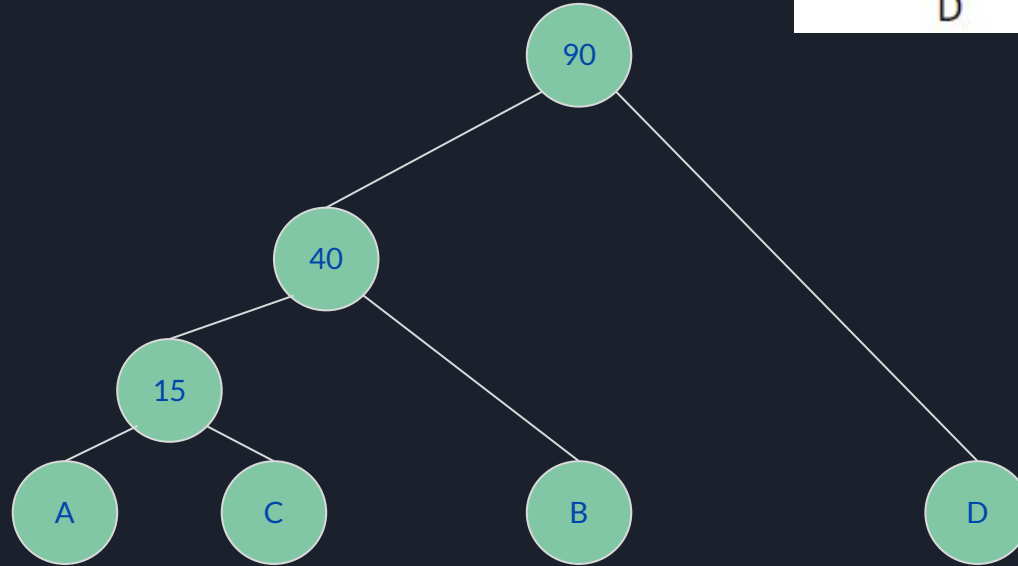
# Huffman Coding

Character	Frequency
A	5
B	25
C	10
D	50



# Huffman Coding

Character	Frequency
A	5
B	25
C	10
D	50



# Huffman Coding

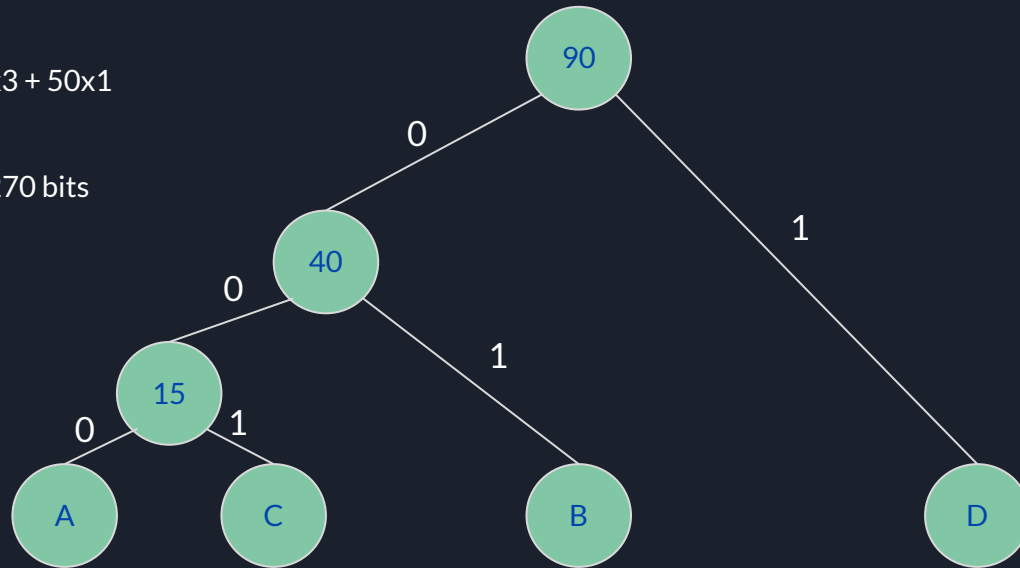
Character	Frequency	Binary Code
A	5	"000"
B	25	"01"
C	10	"001"
D	50	"1"

Total Bits =  $\sum \text{Length of Binary Code (bi)} \times \text{Frequency (fi)}$

=  $5 \times 3 + 25 \times 2 + 10 \times 3 + 50 \times 1$   
= 145 bits

Original Length = 270 bits

**46% Smaller!**





# Huffman Coding

- **Time Complexity:** Getting a minimum element from a Mxn Heap takes  $O(\log N)$  time and the operation is repeated  $N$  times, hence the overall time complexity is  $O(N \log N)$
- **Space Complexity:**  $O(N)$





# Problems in Hashing

- Due to its ever expanding nature, Hash Map's tend to take up a lot of memory.
- Time taken to Hash a key is  $O(N)$ , where  $N$  is the length of the key.
- Larger keys tend to take up a lot of space.



# Combining Huffman Coding and HashMap

## Rationale behind combining these two techniques:

- Huffman Coding excels in compressing data by assigning variable-length codes to input characters based on their frequencies.
- HashMaps, on the other hand, provides a dynamic and efficient indexing mechanism.
- By combining Huffman Coding with Hashing, we can capitalize on the strengths of both techniques. Huffman Coding reduces the overall size of the data, and Hashing ensures a streamlined and rapid access mechanism.

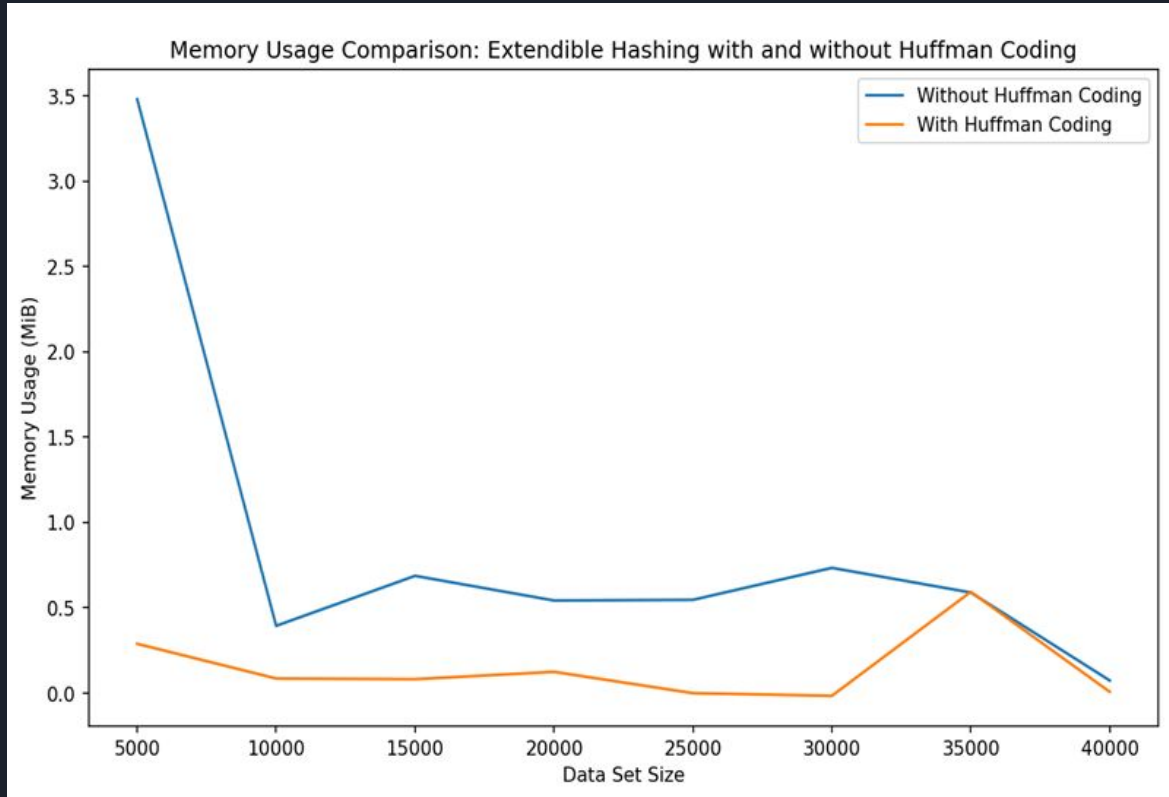


# Huffman Coding and HashMap Working

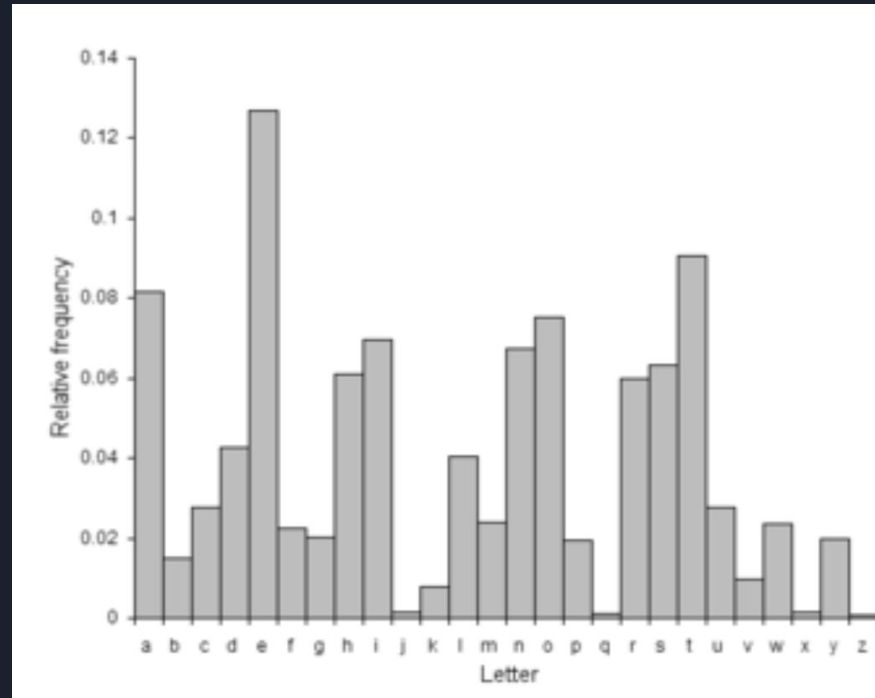
## Algorithm:

- **Input:** A text file with a large amount of repetitive information.
- **Huffman Coding:** Use Huffman Coding to compress the text file. Replace frequently occurring patterns with shorter codes and less frequent patterns with longer codes.
- **Hashing:** Create an hash table to index the location of the compressed file.
- **Indexing:** For each compressed file, compute a hash value and use it to index the hash table. Store the location of the compressed file in the hash table along with the Huffman codes.
- **Decompression:** When decompressing, use the hash table to quickly locate the compressed file and reconstruct the original file using the Huffman codes.

# Huffman Coding and HashMap Results




# Frequency of the English Alphabet





# Potential Use Cases

- Can be used to fasten the file indexing process in Windows which is terribly slow.
- Can be further optimized for Extendible Hashing in Databases to significantly reduce the database size.

A decorative graphic in the top-left corner consisting of two overlapping triangles, one blue and one light green, pointing towards the center.

Thank  
you