Efficient Hashing using Huffman Coding (Dec 2023)

Abstract*— This paper explores the combination of hashing and Huffman coding, two core ideas in computer science, to present a novel method for effective data manipulation. A new paradigm of efficient hashing employing Huffman coding is born from the combination of hashing, which is valued for its quick data retrieval capabilities, and Huffman coding, which is praised for its effectiveness in data compression. The paper guides readers through the reasons behind, methods, and benefits of this integration, emphasizing its possible uses in a variety of computer contexts, including network routing tables, database indexing, and caching systems. This method offers a viable way to improve the effectiveness of different computing applications by reducing collisions, maximizing space consumption, and distributing loads evenly. The difficulties and factors related to dynamic data and implementation overhead are also covered in the document. Overall, this exploration aims to inspire further research and consideration of this hybrid approach in the ever-evolving landscape of computer science.

I. INTRODUCTION

Hashing and Huffman coding are powerful techniques in computer science, each serving distinct purposes. Hashing is commonly employed for fast data retrieval, while Huffman coding is renowned for its efficiency in data compression. This document explores the innovative concept of combining these two methodologies to achieve efficient hashing using Huffman coding.

II. HASHING

Hashing is the process of employing a hash function to map keys and values into a hash table. The purpose is to obtain elements more quickly. The effectiveness of the hash function being utilized determines how efficient the mapping is.

HashMap:

A HashMap is a collection of key-value pairs.

Components of HashMap:

There are majorly four components of hashing:

- **Key:** A **Key** can be any string or integer or an object which is fed as input in the hash function.
- **Value:** A value mapped to the key which we want to store in the hash table.
- **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
- **Hash Table:** A hash table is a data structure that maps keys to values using the hash function.

What is Collision?

There's a chance that two keys will yield the same value because a hash function returns the same hash index. Collisions occur when a newly added key maps to an already occupied position in the hash table; these situations need the application of collision control techniques.

How to handle Collisions?

There are mainly two methods to handle collision:

- Open Addressing
- Separate Chaining

Open Addressing:

Open addressing is a technique for handling collisions, much as independent chaining. Every element in open addressing is kept in the hash table itself. Therefore, the table's size must always be more than or equal to the total number of keys (keep in mind that, if necessary, we can expand the table's size by transferring older data). Another name for this strategy is closed hashing. Probing is the foundation of this entire process. The following forms of probing will become clear to us:

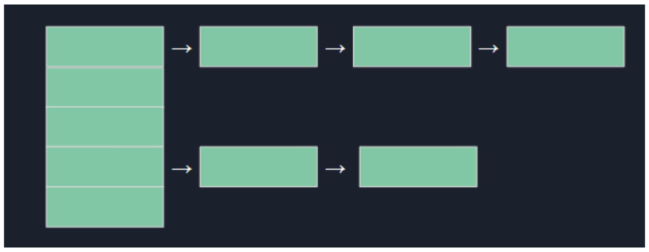
- **Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- **Search(k):** Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- **Delete(k)**: **The delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an

item in a deleted slot, but the search doesn't stop at a deleted slot.

Separate Chaining:
The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques to handle collisions.

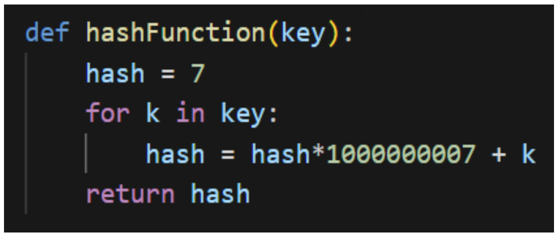
The **linked list** data structure is used to implement this technique. When multiple elements are hashed into the same slot index, then these elements are inserted into a singly linked list which is known as a chain.
Here, all the elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K, then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry, then it means that the entry does not exist.

HashMap using Separating 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:

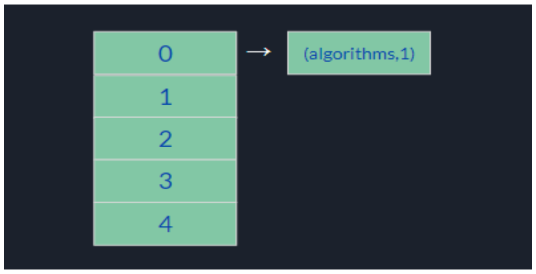


```
def hashFunction(key):
    hash = 7
    for k in key:
        hash = hash*1000000007 + k
    return hash
```

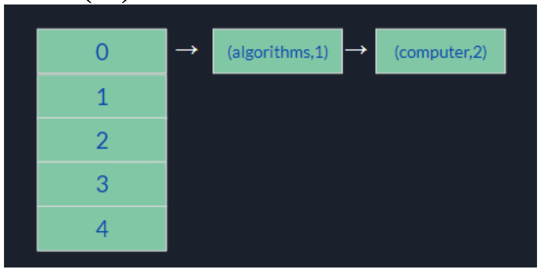
- The index is the hash value generated modulo the size of the array.
- The key-value pair is appended to the linked list at the particular index

Insertion in a HashMap Example:

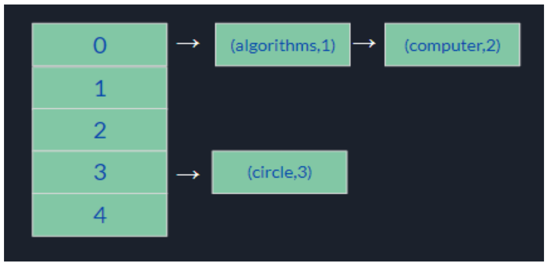
- Insert (algorithms,1): Hash(algorithms) -> 50, Index(50) -> 0



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



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



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 Capacity of the HashMap = Load Factor x 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!

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

Problems in Hashing:

- Due to its ever-expanding nature, HashMap's tend to take up a lot of memory.
- The 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.

III. HUFFMAN CODING

An algorithm for lossless data compression is Huffman coding. The concept is to give input characters variable-length codes, the lengths of which are determined by the frequency of the matching characters. Prefix codes, or variable-length codes assigned to input characters, are bit sequences that are assigned so that the code assigned to one character does not prefix any other character's code. By doing this, Huffman Coding ensures that the produced bitstream may be decoded without any ambiguity.

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 = {a1, a2, a3, …, an} where n is the number of alphabets.

Output: Binary Code B = {b1, b2, b3, …, bn} where bi is a binary code mapped to the alphabet ai.

Objective:

The objective is to create a Huffman Tree based, using the input characters and their frequency, assign them binary codes, and reduce the resultant bit size. The 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"

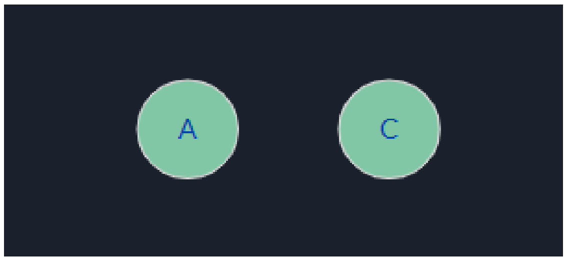
Algorithm to build a Huffman Tree:

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

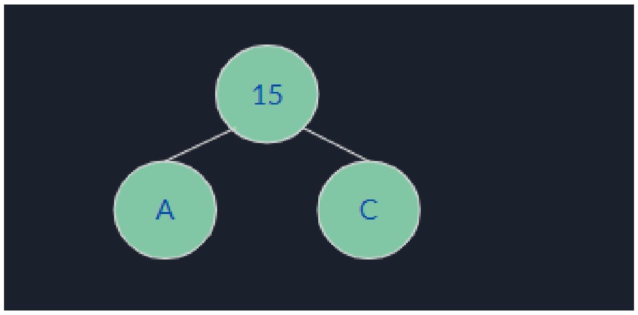
Let us build a Huffman tree for this given table of alphabets and their frequencies.

Character	Frequency
A	5
B	25
C	10
D	50

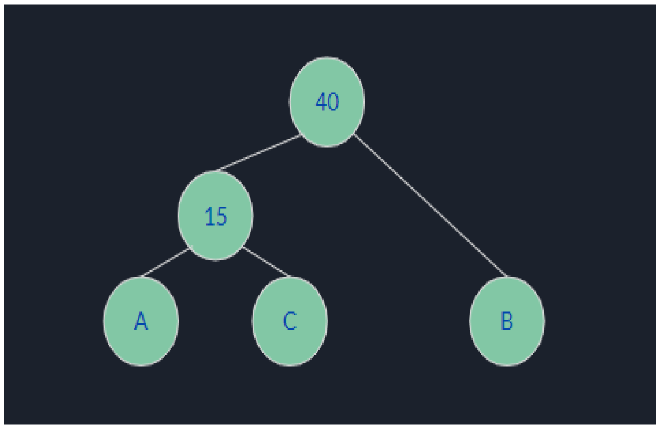
- Remove Two Nodes of Min Frequency from the Min Heap

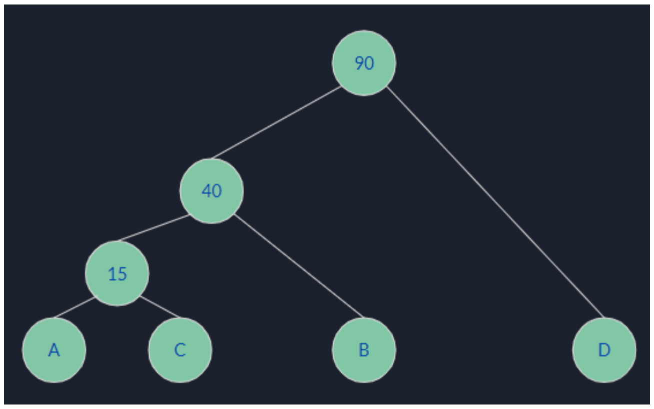


- Create a new internal node with a frequency equal to the sum of its two children and add it to the Min Heap.



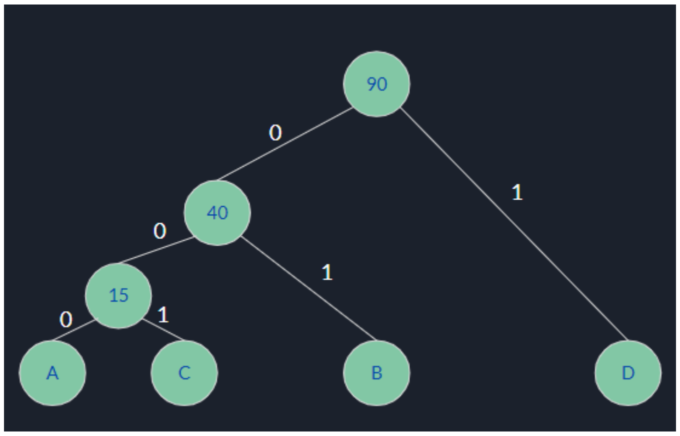
- Repeat the above steps until only one node remains.





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

In the above table we can see the binary representation of the characters. Most frequent character is represented by less number of bits to reduce the memory size and less frequent character is represent by more number of bits.



Total Bits = ∑ Length of Binary Code (bi) x Frequency (fi)

= 5x3 + 25x2 + 10x3 + 50x1
= 145 bits

Original Length = 270 bits (Assuming a 3-bit code for each character)

46% Smaller!

IV. COMBINING HUFFMAN CODING AND HASHING

The 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

Algorithm:

1. Extendible Hashing Initialization:
    - Initialize a global depth D and create an empty directory with 2^D buckets.
    - Each bucket contains a local depth equal to D.
    - Create an empty hash table.
2. Insertion, For each key-value pair (key, value) to be inserted:
    - Compute the hash value using the first D

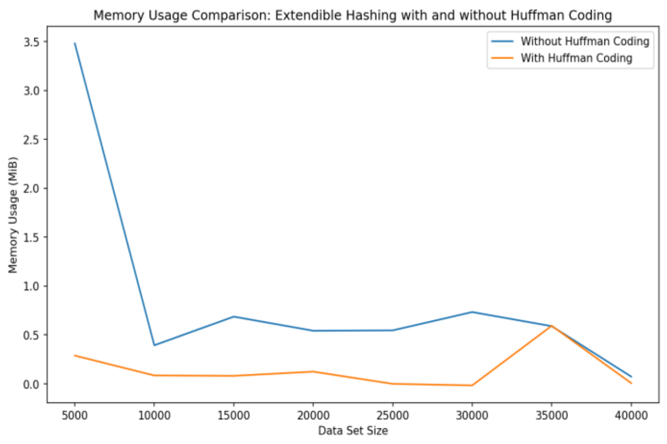
bits of the hash of the key.

- If the local depth of the corresponding bucket is less than D, insert the key-value pair into the bucket.
- If the local depth is equal to D, split the bucket by incrementing its local depth and duplicating the bucket.
- Update the directory to reflect the changes.
- Recompute the hash value for the key with the updated local depth.
- Insert the key-value pair into the appropriate bucket.

3. Huffman Coding for Compression:
   - After each insertion, update the Huffman tree based on the lengths of the compressed codes.
   - Generate Huffman codes for each key in the hash table.
   - Compress the values using the generated Huffman codes.

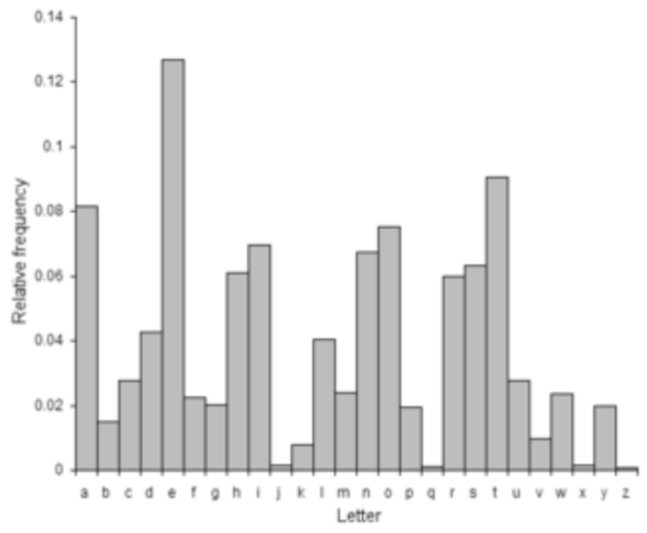
V. Implementation and Results

1. **Classes for Huffman Coding:**
   - **Huffman Node**: Represents a node in the Huffman tree.
   - **HuffmanTree**: Represents the Huffman tree and provides methods for building the tree and generating codes.
2. **Compression and Decompression Functions:**
   - **compress_data**: Uses zlib to compress data.
   - **decompress_data**: Uses zlib to decompress compressed data.
3. **Extendible Hashing Class:**
   - **ExtendibleHashing**: Represents the Extendible Hashing structure. It has methods for initializing, hashing, and inserting data.
4. **Test Functions:**
   - **test_extendible_hashing_without_huffman_and_larger_data**: Inserts words from a larger dataset into an Extendible Hashing structure without Huffman coding.
   - **test_extendible_hashing_with_huffman_and_larger_data**: Inserts words from a larger dataset into an Extendible Hashing structure with Huffman coding.
5. **Plotting Function (plot):**
   - Defines a function to plot the memory usage comparison for Extendible Hashing with and without Huffman coding for different data sizes.
   - Uses the **memory_usage** decorator from **memory_profiler** to measure memory usage during function execution.
   - Iterates over different data sizes, runs the test functions, and records the memory usage.
6. **Data Size Comparison:**
   - The script tests different data sizes (5000, 10000, 15000, ..., 40000).
   - For each data size, it measures the memory usage with and without Huffman coding using the **memory_usage** decorator.
   - The memory usage is recorded as the difference between the final memory usage and the initial memory usage.
7. **Plotting:**
   - The script then uses Matplotlib to create a line plot comparing the memory usage for both cases (with and without Huffman coding) against different data set sizes.
8. **Results:**
   - We have reduced the memory consumption of the HashMap by implementing Huffman Coding



Potential Use Cases:

- Can be used to fasten the file indexing process in Windows which is slow.
- Can be further optimized for Extendible Hashing in Databases to significantly reduce the database size.
- Can be modified to test the performance on the most frequently used characters of the English Alphabet.



VI CONCLUSION

The utilization of Huffman coding inefficient hashing is a novel strategy that combines the advantages of hashing with Huffman coding to tackle issues related to conventional hash functions. This method works well in a variety of applications, such as network routing and databases, by minimizing collisions, maximizing space use, and distributing load evenly. Even though there are obstacles, the concept's advantages in terms of effectiveness and flexibility make it worthwhile for computer scientists to investigate more.