Huffman code example

1. **What is the Greedy Approach?**

The **Greedy approach** is a problem-solving technique that makes the locally optimal choice at each step with the hope of finding the global optimum. In other words, a greedy algorithm chooses the best solution available at each step, without considering the broader problem, and makes decisions based on immediate benefits. This approach is useful in problems where choosing a local optimum leads to a globally optimal solution.

Key characteristics of the Greedy approach:

- It works in a top-down manner.

- It is a **heuristic** approach, meaning it does not guarantee an optimal solution for all problems.

- It focuses on the current step, without considering the consequences of future steps.

Example: In the **Fractional Knapsack problem**, you take as much as possible of the highest value-to-weight ratio item until the knapsack is full. This approach works because you can take fractional parts of the items.

2. **Explain Huffman Coding with Example Using the Greedy Approach**

Huffman Coding is a **greedy algorithm** used for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. This method works by building a **Huffman Tree** using the greedy approach, which makes decisions based on the least frequent characters first and combines them iteratively to create a final tree.

Steps in Huffman Coding (Greedy Approach):

1. **Frequency Analysis**: Calculate the frequency of each character in the input string.

2. **Create Nodes**: Create a leaf node for each character, where each node contains the character and its frequency.

3. **Build the Huffman Tree**:

   - Place all the nodes in a **min-heap** (or priority queue) based on their frequencies (lower frequencies have higher priority).

   - Repeatedly extract the two nodes with the lowest frequencies from the heap.

   - Create a new internal node whose frequency is the sum of the two nodes. This new node becomes the parent of the two extracted nodes.

   - Insert the new node back into the heap.

   - Repeat the process until only one node remains in the heap, which is the root of the Huffman Tree.

4. **Assign Codes**: Traverse the tree from root to leaves. Assign `0` for left edges and `1` for right edges. The Huffman code for each character is the path from the root to that character.

Example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

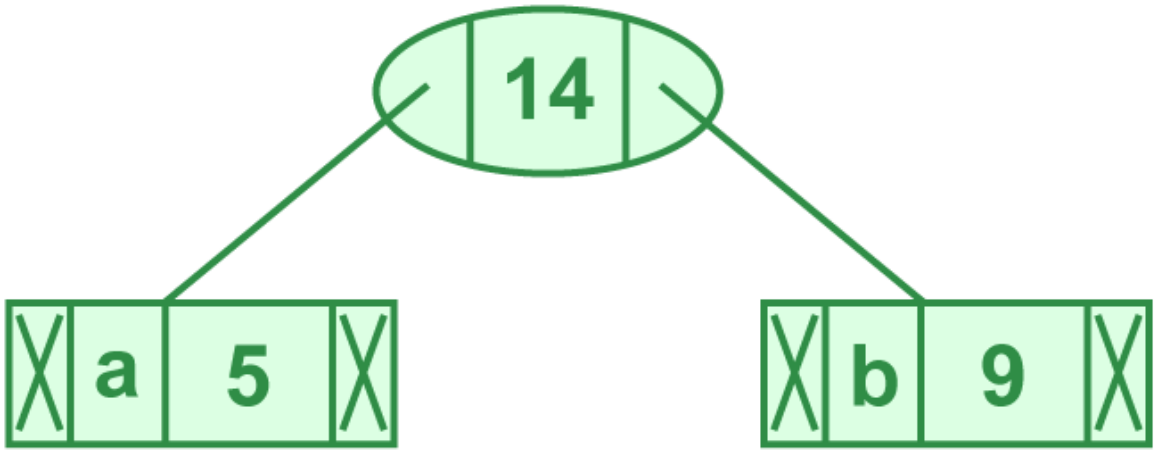


Illustration of step 2

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

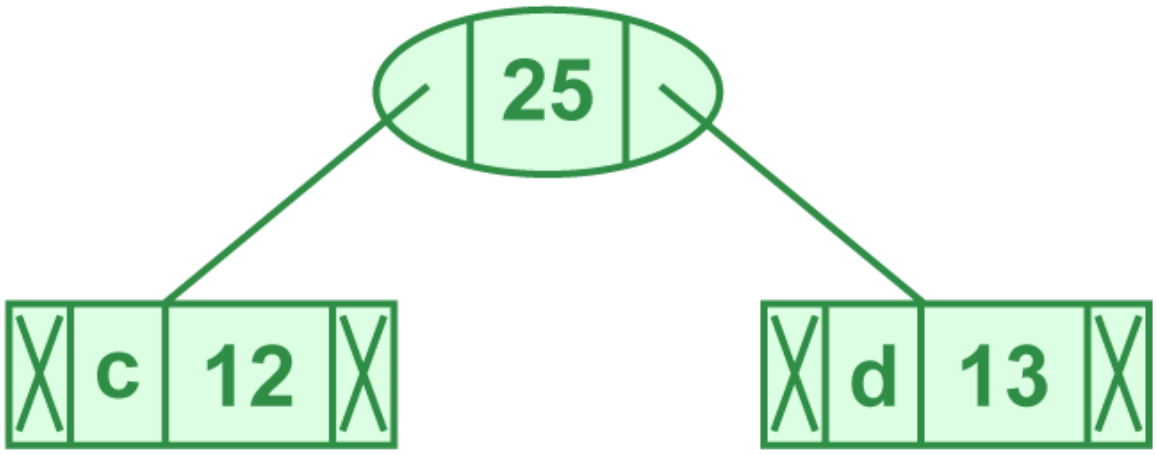


Illustration of step 3

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character Frequency

Internal Node 14

e 16

Internal Node 25

f 45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

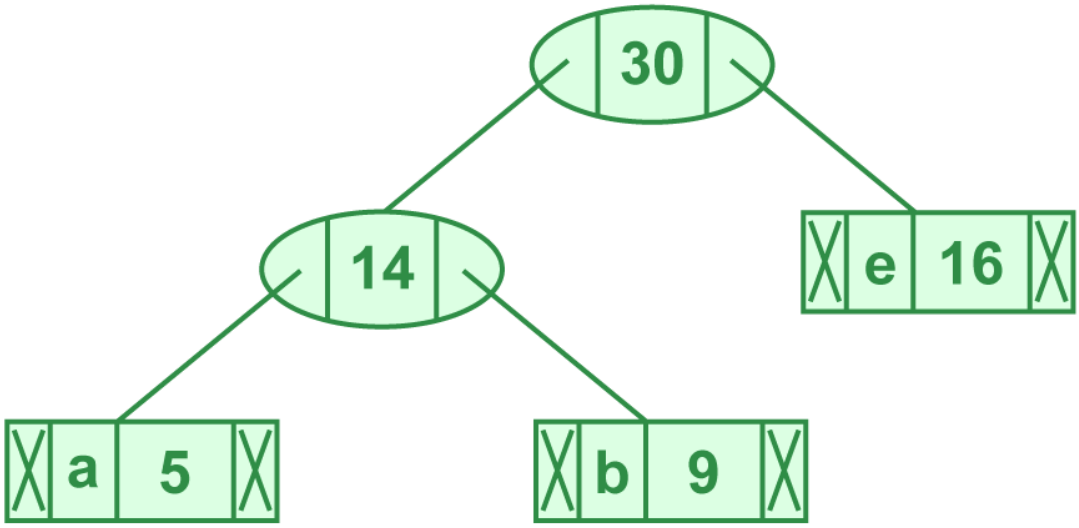


Illustration of step 4

Now min heap contains 3 nodes.

character Frequency

Internal Node 25

Internal Node 30

f 45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

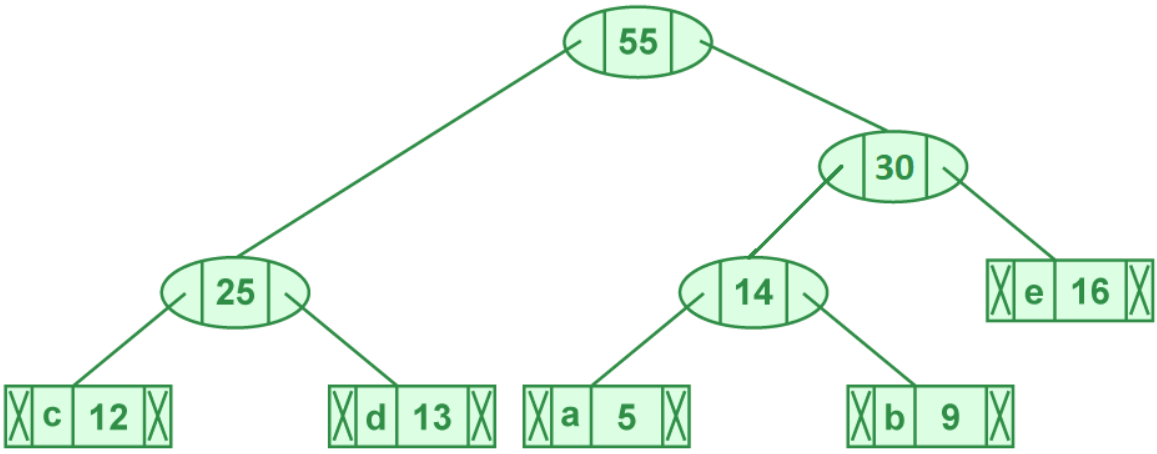


Illustration of step 5

Now min heap contains 2 nodes.

character Frequency

f 45

Internal Node 55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

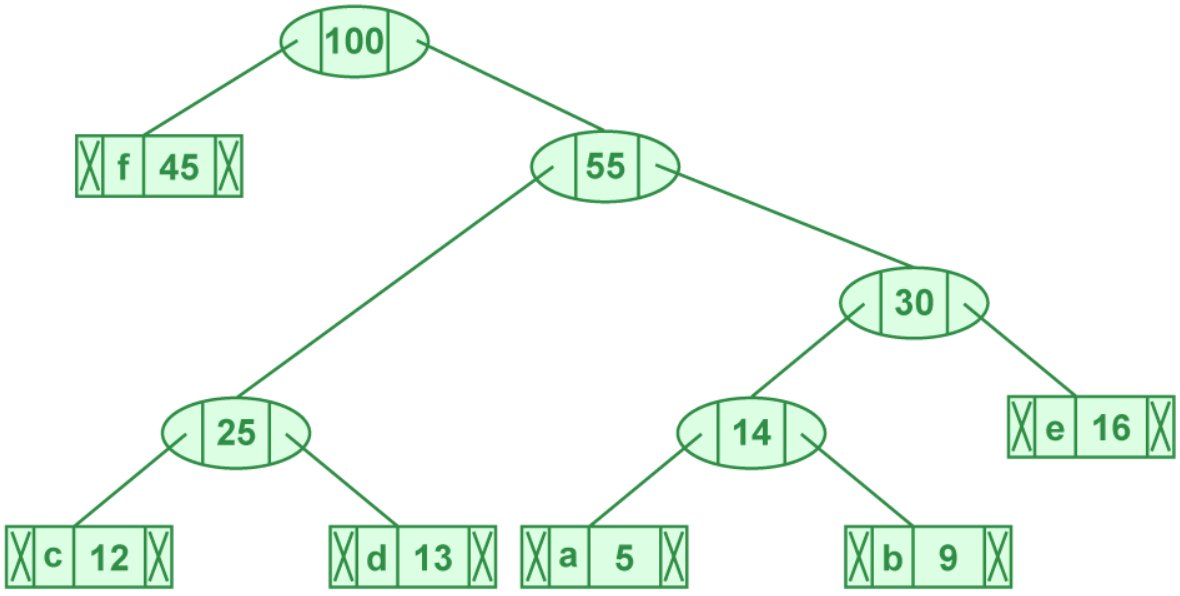


Illustration of step 6

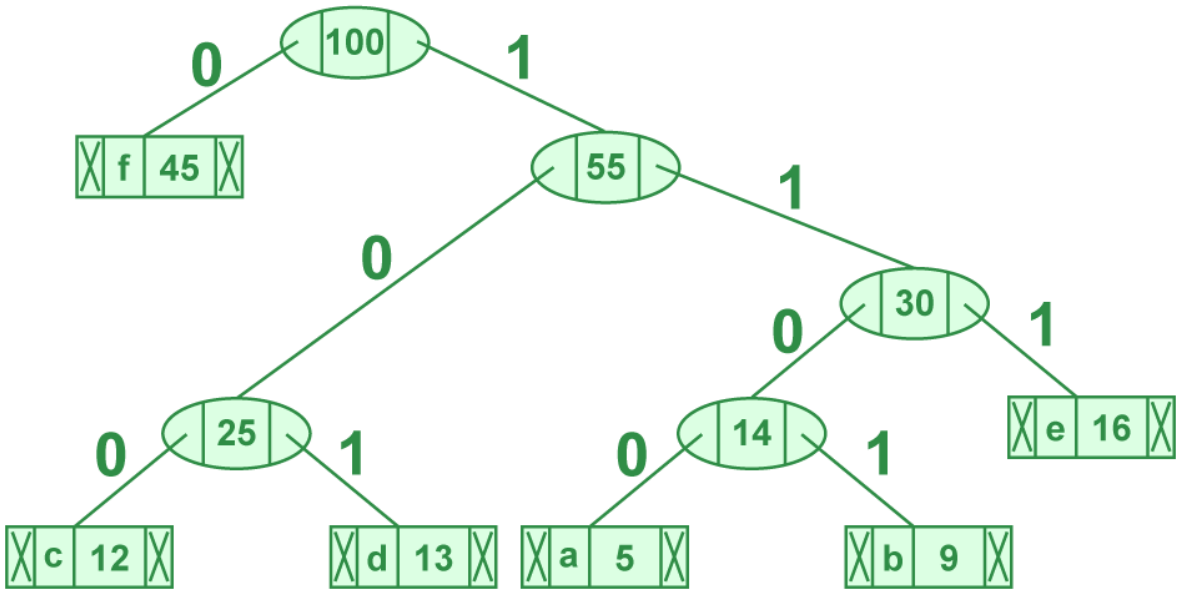
Now min heap contains only one node.

character Frequency

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



Steps to print code from HuffmanTree

The codes are as follows:

character code-word

f	0
c	100
d	101
a	1100
b	1101
e	111

3. **Analyze Time Complexity of Huffman Coding**

The **time complexity** of the Huffman coding algorithm depends primarily on the operations involved in building the Huffman tree.

- **Step 1 (Frequency Calculation)**: This step involves counting the frequency of each character, which takes **O(n)** time, where `n` is the number of characters in the input string.

- **Step 2 (Building the Huffman Tree)**:

  - Constructing the min-heap (priority queue) initially takes **O(m log m)** time, where `m` is the number of unique characters (or nodes).

  - Each extraction and insertion operation on the heap takes **O(log m)** time, and since we perform `m - 1` extractions and insertions (each combining two nodes), the total time for building the tree is **O(m log m)**.

- **Step 3 (Generating Codes)**: This step involves traversing the Huffman tree, which takes **O(m)** time, where `m` is the number of unique characters.

Overall Time Complexity:

- The overall time complexity of Huffman coding is dominated by the heap operations, so it is:

  - **O(m log m)**, where `m` is the number of unique characters in the input.

  - In the worst case, `m` can be equal to `n` (the number of characters in the input string), so the worst-case time complexity is **O(n log n)**.

Space Complexity:

- The space complexity is determined by the storage required for the input string, the frequency map, and the Huffman tree. The space complexity is **O(m)**, where `m` is the number of unique characters in the input.

Summary:

1. The **Greedy approach** in Huffman coding makes locally optimal choices by combining the least frequent characters first to construct the optimal Huffman tree.

2. **Huffman coding** is used to compress data by assigning shorter codes to more frequent characters, thus minimizing the total number of bits used to encode the data.

3. The **time complexity** of Huffman coding is **O(n log n)**, where `n` is the number of characters, and the **space complexity** is **O(m)**, where `m` is the number of unique characters.

……Code Explanation……

Let's break down the **Huffman coding** code step by step, explaining each part in detail:

1. **Libraries Used**

```cpp
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>
```

- **`#include <iostream>`**: This is used for input and output operations (e.g., `cout`, `cin`).
- **`#include <queue>`**: This includes the `priority_queue` data structure. In this case, it's used for

building the **min-heap**, which is central to the Huffman coding algorithm.

- **`#include <unordered_map>`**: This is used for creating a frequency map (a hash map) where each character is associated with its frequency.

- **`#include <vector>`**: Although not directly used here, this is required for certain types of containers in C++.

---

2. **Node Structure**

```cpp
struct Node {
    char ch;
    int freq;
    Node *left, *right;

    Node(char character, int frequency) {
        ch = character;
```

```
        freq = frequency;
        left = right = nullptr;
    }
};
```

- **`Node`**: This is the structure that represents each node in the Huffman tree. It has:
  - **`ch`**: The character stored at this node.
  - **`freq`**: The frequency of that character.
  - **`left` and `right`**: Pointers to the left and right children of the node (used to build the binary tree structure).
  - The constructor initializes these values when a new `Node` is created.

---

3. **Comparison Class for Min-Heap**

```cpp
struct Compare {
```

```
    bool operator()(Node* left, Node* right) {
        return left->freq > right->freq;
    }
};
```

- **`Compare`**: This is a custom comparator used to order the `Node` pointers in the **min-heap** (priority queue).
  - The priority queue will store `Node*` pointers.
  - The comparator `operator()` returns `true` if the frequency of `left` is greater than `right`, meaning the min-heap will prioritize the node with the lower frequency.
  - This ensures that the **least frequent nodes** are always at the top of the heap, which is essential for building the Huffman tree.

---

4. **Print Huffman Codes**

```cpp
void printCodes(Node* root, string str) {
    if (!root)
        return;

    if (!root->left && !root->right) {
        cout << root->ch << ": " << str << endl;
    }

    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}
```

- **`printCodes`**: This function recursively traverses the Huffman tree and prints the Huffman codes for each character.
  - **Base Case**: If the `root` is `nullptr`, it returns immediately.

- If the current node is a **leaf node** (it has no children), it prints the character and the corresponding code (`str`).

- Otherwise, the function recursively explores both the left and right subtrees:

  - It adds `"0"` to `str` when moving left.

  - It adds `"1"` to `str` when moving right.

---

5. **Building the Huffman Tree**

```cpp
void buildHuffmanTree(unordered_map<char, int> &freqMap) {
    priority_queue<Node*, vector<Node*>, Compare> minHeap;

    for (auto pair : freqMap) {
        minHeap.push(new Node(pair.first, pair.second));
    }
```

```
    while (minHeap.size() != 1) {
        Node *left = minHeap.top();
        minHeap.pop();
        Node *right = minHeap.top();
        minHeap.pop();

        int sum = left->freq + right->freq;
        Node *newNode = new Node('\0', sum);
        newNode->left = left;
        newNode->right = right;

        minHeap.push(newNode);
    }

    Node* root = minHeap.top();

    printCodes(root, "");
}
```

```

- **`buildHuffmanTree`**: This function constructs the Huffman tree using the provided character frequencies stored in the `freqMap`.

1. **Creating a Min-Heap**:

   - The function creates a **min-heap** (priority queue), where the nodes are ordered based on their frequencies, thanks to the `Compare` comparator.

   - For each character-frequency pair in `freqMap`, it creates a new `Node` and pushes it into the heap. This initializes the heap with the leaf nodes of the Huffman tree.

2. **Building the Tree**:

   - The core part of the Huffman algorithm starts. While there is more than one node in the heap (i.e., the heap size is not 1), it performs the following steps:

     - **Extract two nodes** with the smallest frequencies from the heap using `minHeap.top()` (i.e., the least frequent nodes).
```

- **Create a new internal node**: The frequency of the new node is the sum of the frequencies of the two extracted nodes. This node has no character (`'\0'`).

- **Set the extracted nodes as children**: The two extracted nodes become the left and right children of the new node.

- **Push the new node back into the heap**.

- This process continues until only one node remains in the heap, which becomes the **root** of the Huffman tree.

3. **Print the Codes**:

- Once the tree is built, the `printCodes` function is called to output the Huffman codes for each character.

---

6. **Main Function**

```cpp
int main() {
```

```
int n;
unordered_map<char, int> freqMap;

cout << "Enter the number of characters: ";
cin >> n;

for (int i = 0; i < n; i++) {
    char ch;
    int freq;
    cout << "Enter character:" << endl;
    cin >> ch;
    cout<<"enter the frequency:"<<endl;
    cin>>freq;
    freqMap[ch] = freq;
}

buildHuffmanTree(freqMap);

return 0;
```

}
```

- **`main`**: This is the entry point of the program.
  - **Input**:
    - The user is asked to enter the number of characters (`n`).
    - For each character, the user is prompted to enter the character itself (`ch`) and its frequency (`freq`).
    - These pairs are stored in the `freqMap` (an unordered map), which will be used to build the Huffman tree.
  - **Huffman Tree Construction**: After collecting the input, the `buildHuffmanTree` function is called to create the tree and print the codes.

---

### Example Walkthrough

Let’s walk through an example. Suppose the user enters:
```

Enter the number of characters: 5

Enter character: A enter the frequency: 5

Enter character: B enter the frequency: 9

Enter character: C enter the frequency: 12

Enter character: D enter the frequency: 13

Enter character: E enter the frequency: 16

```

1. **Create Initial Nodes**:

   ```

   A: 5, B: 9, C: 12, D: 13, E: 16

   ```

2. **Build the Min-Heap**:

   - Initially: `A:5`, `B:9`, `C:12`, `D:13`, `E:16` are inserted into the heap.

   - Nodes with smallest frequencies are combined iteratively:

     - Combine A (5) and B (9) → New node: AB (14)
   ```

- Combine C (12) and D (13) → New node: CD (25)

- Combine AB (14) and E (16) → New node: ABE (30)

- Combine CD (25) and ABE (30) → Final node: CDABE (55)

3. **Traverse the Huffman Tree and Assign Codes**:

```
        (55)
       /    \
     (25)    (30)
    /   \    /   \
  (12)  (13) (14)  (16)
  C     D    AB    E
```

- The codes:

  - A: 01

  - B: 00

  - C: 10

  - D: 11

- E: 111

The output will be:

```
B: 00

A: 01

C: 10

D: 11

E: 111
```

Conclusion:

This code efficiently builds a **Huffman tree** and generates Huffman codes using a **greedy algorithm**. It stores frequencies in a **priority queue (min-heap)** and constructs the tree by combining the least frequent nodes. The program's output shows the optimal encoding for the characters.

……Applications……..

Huffman coding is widely used in various applications due to its efficiency in compressing data without losing any information (lossless compression). Below are some key **applications of Huffman coding**:

1. **Data Compression**

- **File Compression**: Huffman coding is a fundamental technique in file compression algorithms like **ZIP** and **GZIP**. It helps reduce the file size by encoding frequent characters with shorter codes and less frequent characters with longer codes.

- **Image Compression**: It is often used in image compression formats like **JPEG**, where it is applied to the **DCT coefficients** (Discrete Cosine Transform) after transforming the image from spatial to frequency domain.

- **Text Compression**: Huffman coding is used in text compression algorithms such as **LZW** (Lempel-Ziv-Welch) and **Deflate**. It helps compress large text files efficiently.

2. **Data Transmission**

- **Efficient Transmission**: Huffman coding is used in communication systems to reduce the amount of data being transmitted over networks, especially when bandwidth is limited. For example, it helps in efficient **telecommunication** or **satellite communications** where minimizing the data sent is crucial.

- **Error-Free Transmission**: Huffman coding is also used in **data link layer protocols** to compress data before transmission, helping reduce errors or the chance of data loss during transmission.

3. **Multimedia Applications**

- **Video Compression**: In video codecs like **H.264** and **HEVC (High Efficiency Video Coding)**, Huffman coding is used to compress video data by encoding frequently occurring patterns with shorter bit codes, improving storage and streaming efficiency.

- **Audio Compression**: It is also used in audio codecs like **MP3** and **AAC** to compress sound files by encoding frequent audio patterns more

efficiently, reducing file size while maintaining audio quality.

4. **File Formats and Standards**

- **JPEG Image Compression**: Huffman coding is applied in JPEG image compression to compress image data after the Discrete Cosine Transform (DCT) is applied, leading to smaller file sizes.

- **PNG (Portable Network Graphics)**: Although PNG uses **DEFLATE** (which is based on LZ77 and Huffman coding), Huffman coding is a key part of its lossless compression mechanism.

- **PDF (Portable Document Format)**: Some PDF files use Huffman coding for compressing text, images, and other data types, which helps reduce the file size while keeping it suitable for fast transmission or download.

5. **Data Storage**

- **Database Indexing**: Huffman coding is used in database systems to efficiently store and retrieve data by compressing large text fields, such as indexes or logs.

- **Efficient Disk Storage**: Some disk storage systems use Huffman coding to reduce the storage space required for certain files, allowing for higher storage capacity without compromising the integrity of the data.

6. **Barcode Systems**

- **QR Codes and Barcodes**: Huffman coding is applied in **QR codes** and other barcode systems to efficiently store data in a compact manner while ensuring that data can be easily decoded by scanners.

7. **Web Technologies**

- **Web Page Compression (GZIP)**: Huffman coding is a part of the **GZIP** compression algorithm used in web technologies to compress web pages. When a website is accessed, the server can send compressed content (HTML, CSS, JavaScript) to the browser to reduce loading time and improve page performance.

- **Content Delivery Networks (CDNs)**: CDNs use Huffman coding to optimize the delivery of compressed content to users globally, reducing latency and improving user experience.

8. **Lossless Data Coding in Cryptography**

- **Data Encryption**: Huffman coding can be combined with encryption algorithms in cryptographic systems to reduce the size of the encrypted message or file. In such cases, Huffman coding can help to compress the ciphertext without compromising the security of the data.

9. **DNA and Genomic Data Compression**

- **Bioinformatics**: Huffman coding is used in the compression of DNA and genomic sequences, where certain nucleotide sequences occur more frequently. This helps in reducing the storage and transmission requirements for large biological data sets.

10. **Speech Recognition and Processing**

- **Speech Compression**: Huffman coding is used in some **speech codecs** to compress audio signals by encoding repetitive patterns in speech. This is common in **voice-over-IP (VoIP)** applications, reducing

bandwidth usage while maintaining acceptable audio quality.

11. **Medical Imaging**

- **DICOM (Digital Imaging and Communications in Medicine)**: Huffman coding is used in medical image formats like **DICOM** for the compression of medical images like MRIs, CT scans, and X-rays. This allows medical professionals to store and transmit high-resolution images in a more efficient manner.

Summary

In summary, Huffman coding is a crucial technique for **lossless data compression**, widely used in applications ranging from **file compression**, **image and video compression**, and **telecommunications** to **bioinformatics** and **cryptography**. Its efficiency lies in encoding frequent patterns with shorter bit codes, enabling substantial reductions in data size while preserving all original information.