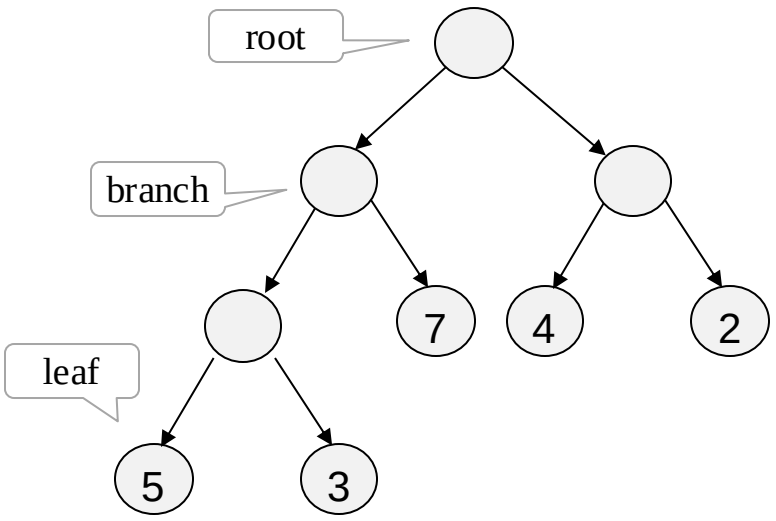
Lecture 12-2

Trees

Tree abstraction



Depth First Search algorithm (DFS)

```
// Prints the leaves, left to right
print(node):
  if node.isLeaf
     print(node)
  else
     print(node.left)
     print(node.right)
```

output: `5 3 7 4 2`

Binary tree

- A data structure consisting of labeled *nodes* and *edges*
- Each node has at most 2 children, and at most 1 parent
- One node has no parent. It's called the *root*
- Nodes with no children are called *leaves* (or *terminal*)
- All the other nodes are called *branches* (or *interim*)

Notes

A binary tree is a recursive data structure:

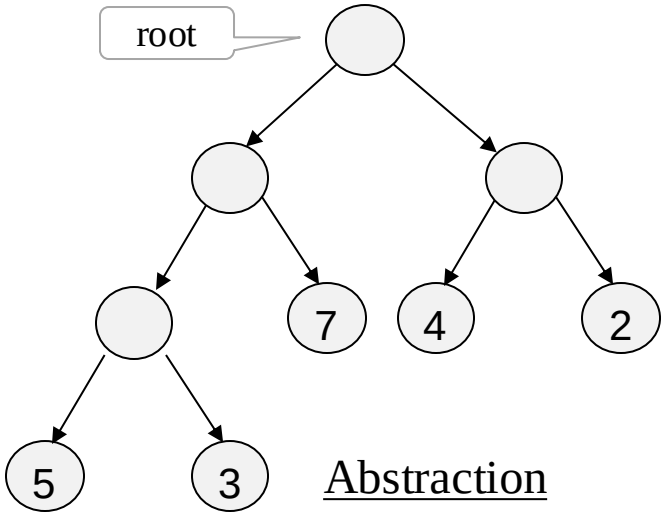
The children of each node are themselves binary trees

To process trees, we use recursive algorithms.

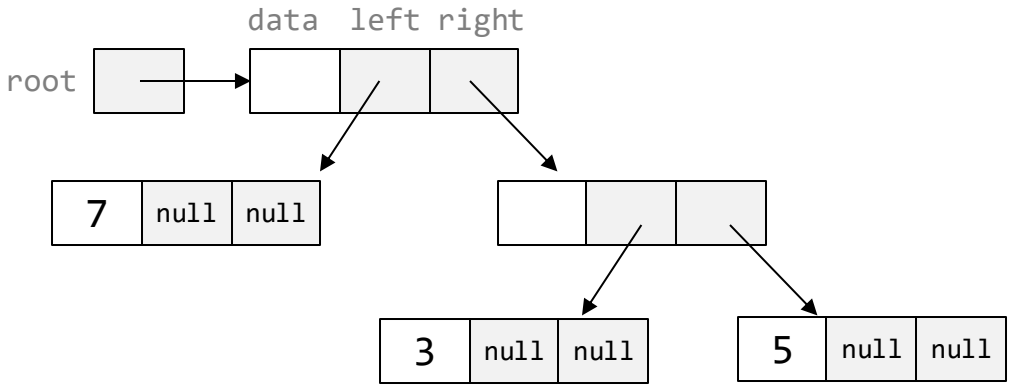
Tree processing example

Print all the leaves, left to right.

Tree implementation



Abstraction

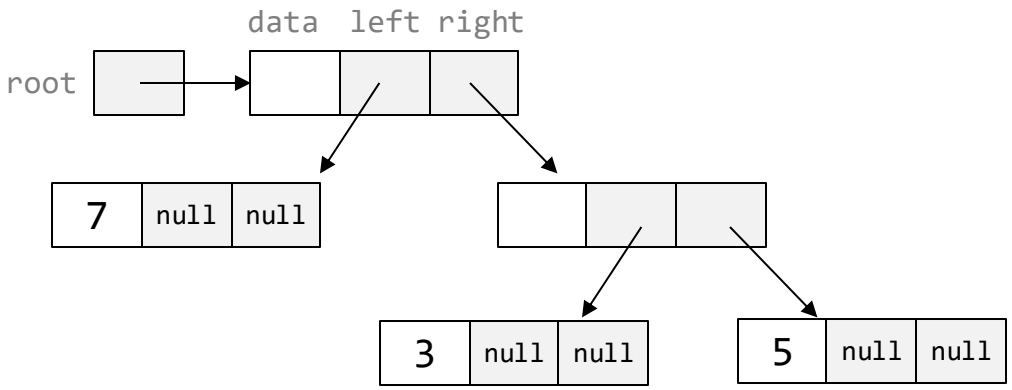


Implementation (different tree)

Connected `Node` objects

Tree implementation

```
public class Node {
    int data;
    Node left, right;

    // Constructs a branch node
    Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;  this.right = right;
    }

    // Constructs a leaf node
    Node(int data) {
        this.data = data;
        this.left = null;  this.right = null;
    }

    // Checks if this node is a leaf node
    boolean isLeaf() {
        return ((left == null) &&
                (right == null));
    }

    // Prints all the leaves below this node, L to R
    public void print() {
        if (isLeaf()) {
            System.out.print(data + " ");
        } else {
            left.print();
            right.print();
        }
    }
}
```



Client code:

```
// Constructs the above tree
Node leaf1 = new Node(7);
Node leaf2 = new Node(3);
Node leaf3 = new Node(5);
Node branch = new Node(0, leaf2, leaf3);
Node root = new Node(0, leaf1, branch);
root.print();
```

output:

```
7 3 5
```

Tree implementation

```
public class Node {

    int data;
    Node left, right;

    // Constructs a branch node
    Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;  this.right = right;
    }

    // Constructs a leaf node
    Node(int data) {
        this.data = data;
        this.left = null;  this.right = null;
    }

    // Checks if this node is a leaf node
    boolean isLeaf() {
        return ((left == null) &&
                (right == null));
    }

    // Prints all the leaves below this node, L to R
    public void print() {
        if (isLeaf()) {
            System.out.print(data + " ");
        } else {
            left.print();
            right.print();
        }
    }
}
```

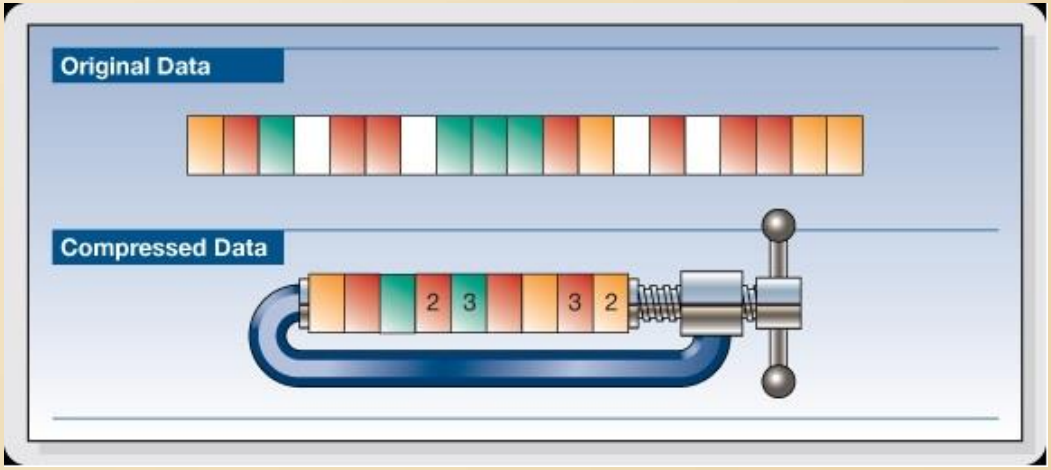
Next

We will now turn to describe an elegant and important application that uses trees:

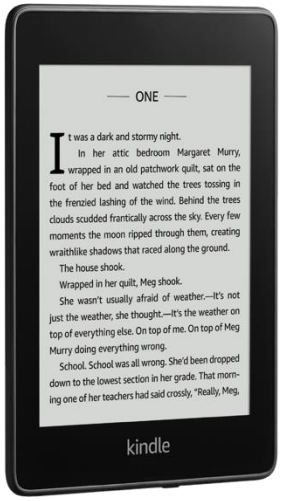
Compression

Lecture 12-2

Compression

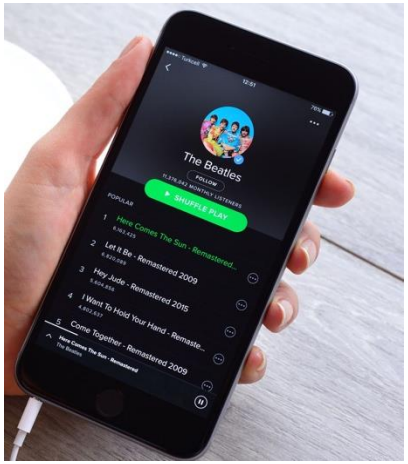


Data explosion



text

video



audio

Every day, 10^{19} new bits are added to the Internet

Goals

- Minimize storage space
- Minimize download / upload time

Compression algorithms

- Lossless (`ZIP`, `PNG`, …)
- Lossy (`JPG`, `MPEG`, `MP3`, …)

Data explosion

Example (*A Promised Land*, by Barack Obama, 738 pages)

… I began writing this book shortly after the end of my presidency—after Michelle and I had boarded Air Force One for the last time and traveled west for a long-deferred break. The mood on the plane was bittersweet. Both of us were drained, physically and emotionally, not only by the labors of the previous eight years but by the unexpected results of an election in which someone diametrically opposed to everything we stood for had been chosen as my successor. Still, having run our leg of the race to completion, we took satisfaction in knowing that we'd done our very best—and that however much I'd fallen short as president, whatever projects I'd hoped but failed to accomplish, the country was in better shape now than it had been when I'd started. For a month, Michelle and I slept late, ate leisurely dinners, went for long walks, swam in the ocean, took stock, replenished our friendship, rediscovered our love, and planned for a less eventful but hopefully no less satisfying second act. …

The full book:

2,300,000 characters × 16 bits =
36,864,000 bits;

Can we cut down the number of bits by about 90%, without losing any contents?

Yes we can!

Lecture plan

- Trees
- Compression: Algorithms
- Compression: Implementation
- Compression: Extensions

Character frequency

Example (this paragraph only)

… I began writing this book shortly after the end of my presidency—after Michelle and I had boarded Air Force One for the last time and traveled west for a long-deferred break. The mood on the plane was bittersweet. Both of us were drained, physically and emotionally, not only by the labors of the previous eight years but by the unexpected results of an election in which someone diametrically opposed to everything we stood for had been chosen as my successor. Still, having run our leg of the race to completion, we took satisfaction in knowing that we'd done our very best—and that however much I'd fallen short as president, whatever projects I'd hoped but failed to accomplish, the country was in better shape now than it had been when I'd started. For a month, Michelle and I slept late, ate leisurely dinners, went for long walks, swam in the ocean, took stock, replenished our friendship, rediscovered our love, and planned for a less eventful but hopefully no less satisfying second act. …

text analysis

frequencies

0	
...	
'a'	54
'b'	16
'c'	25
'd'	39
'e'	107
'f'	22
'g'	10
'h'	40
...	
255	

Character frequency

Example (small data, for testing)

```
// Builds a frequencies array, for testing
double[] freq = new double[256];
freq['a'] = 8; freq['b'] = 3;
freq['c'] = 1; freq['d'] = 1; freq['e'] = 1;
freq['f'] = 1; freq['g'] = 1; freq['h'] = 1;
```

text analysis

frequencies

0	
...	
'a'	8
'b'	3
'c'	1
'd'	1
'e'	1
'f'	1
'g'	1
'h'	1
...	
255	

Reminder: `char` values are numbers, e.g. `'a'` = 97 and `'h'` = 104

(Example adapted from *Structure and Interpretation of Computer Programs* by Abelson and Sussman, MIT Press)

Building a Huffman tree

Example (small data, for testing)

```
// Builds a frequencies array, for testing
double[] freq = new double[256];
freq['a'] = 8; freq['b'] = 3;
freq['c'] = 1; freq['d'] = 1; freq['e'] = 1;
freq['f'] = 1; freq['g'] = 1; freq['h'] = 1;
```

text analysis

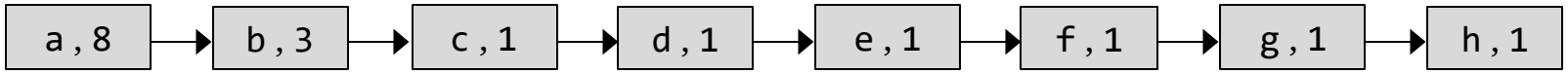
frequencies

0	
...	
'a'	8
'b'	3
'c'	1
'd'	1
'e'	1
'f'	1
'g'	1
'h'	1
...	
255	

First step in building a *Huffman tree*:

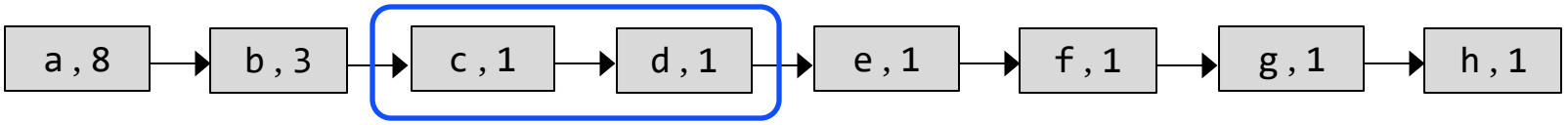
Construct a list from the frequencies array:

Reminder: `char` values are numbers, e.g. `'a'` = 97 and `'h'` = 104

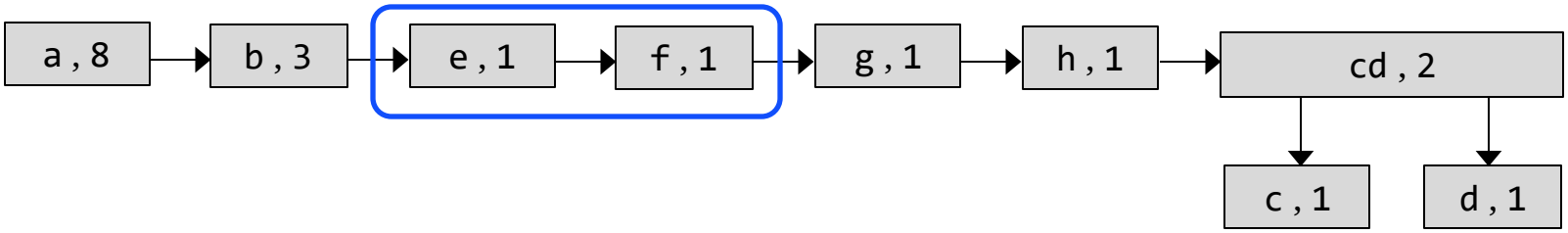


Building a Huffman tree

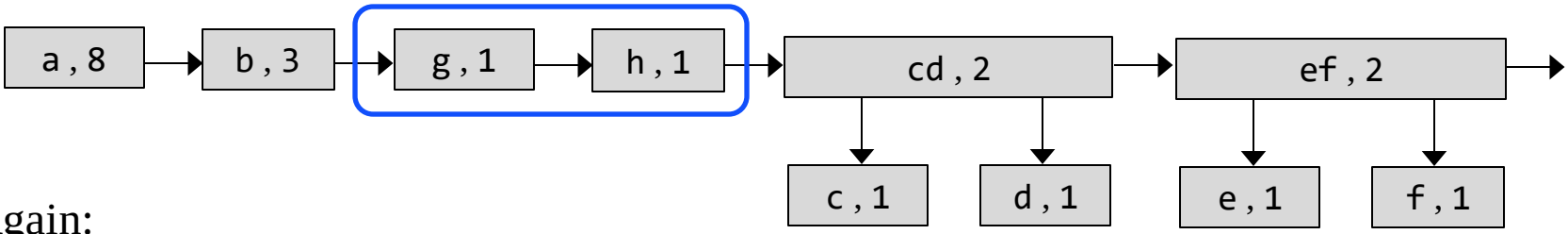
Construct a list from the frequencies array:



Remove the first two nodes that have minimal frequencies, combine them, and add the combined node to the list:



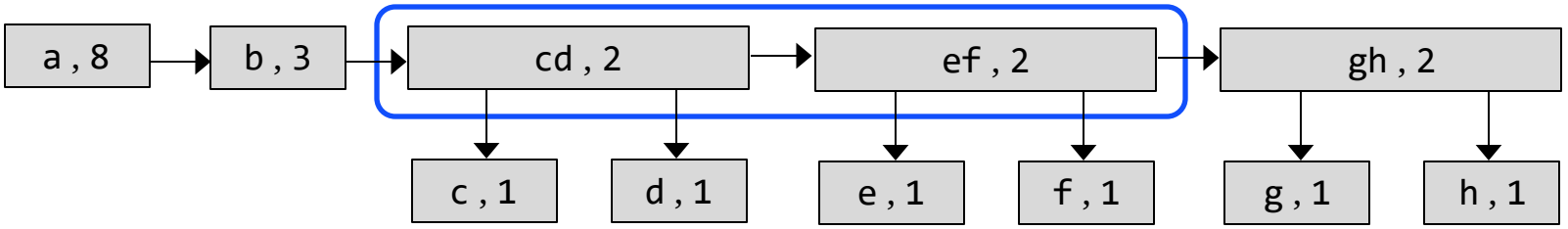
Again:



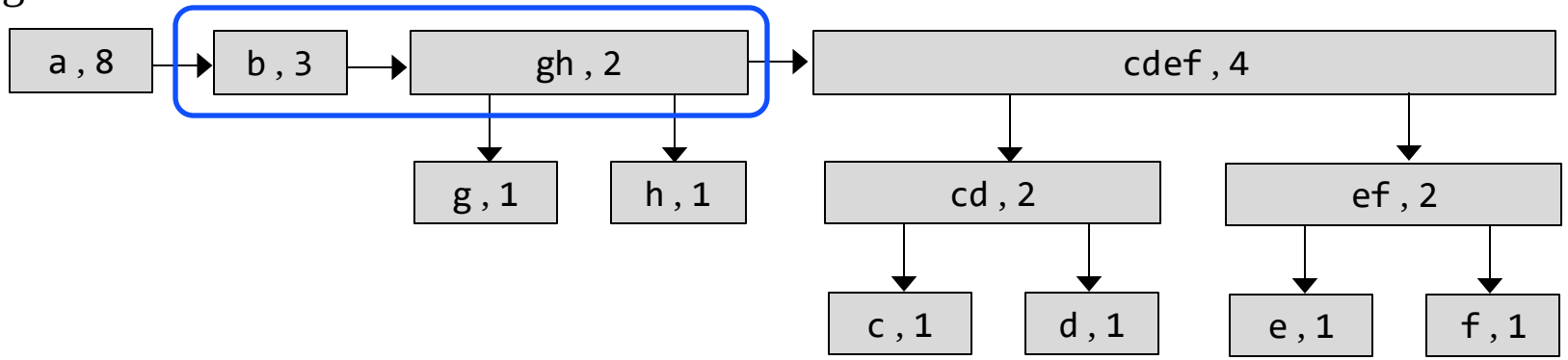
Again:

Building a Huffman tree

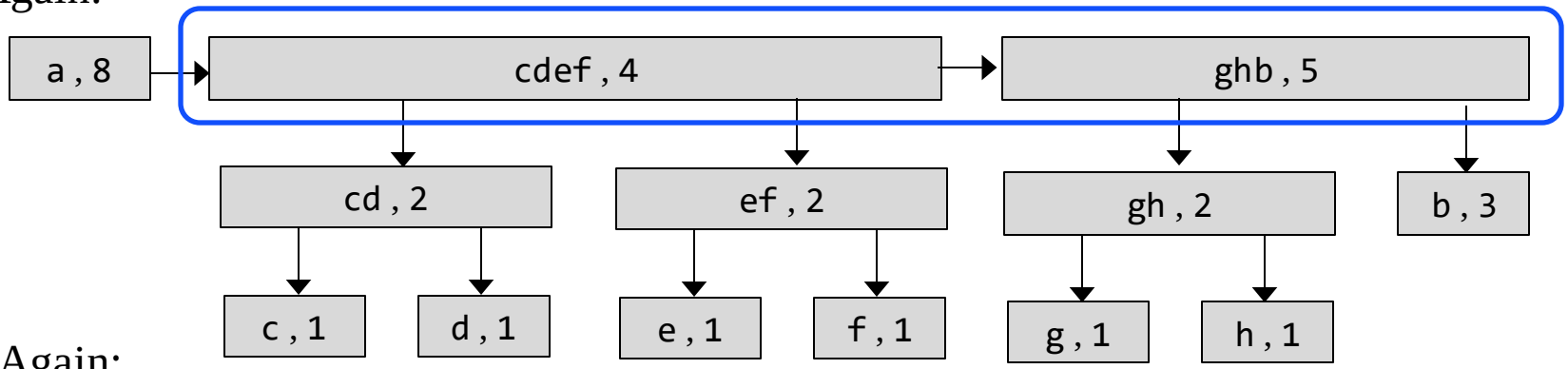
Continue from previous slide:



Again:



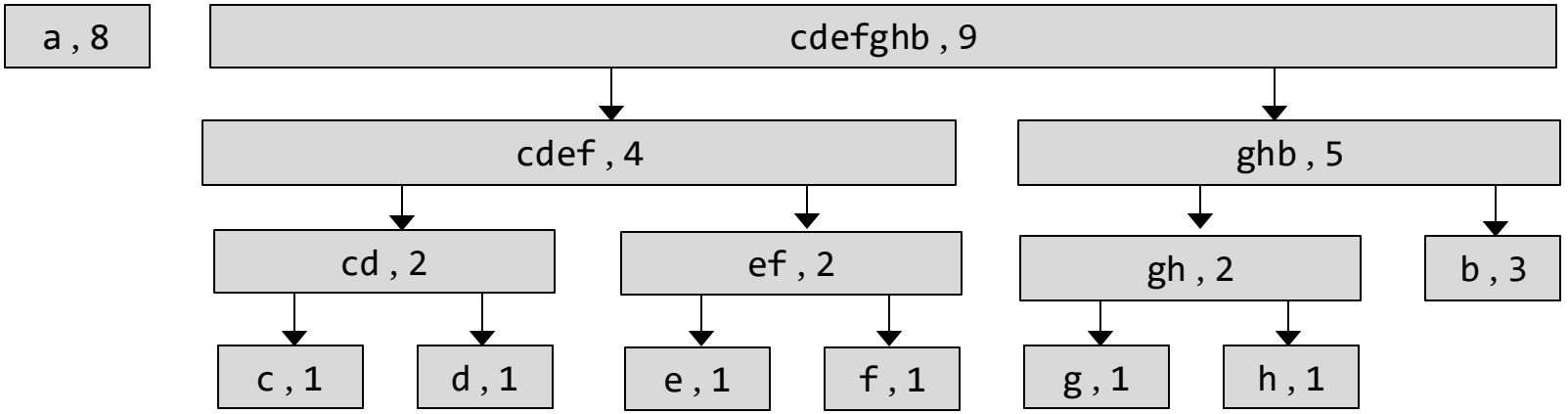
Again:



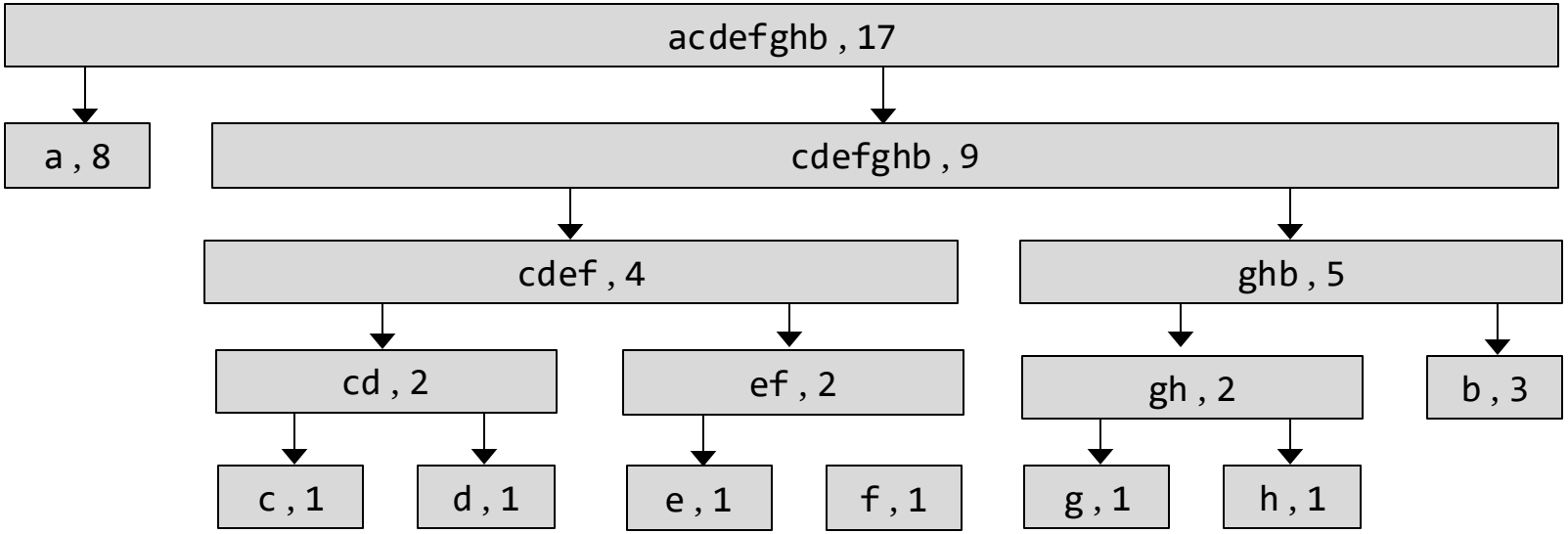
Again:

Building a Huffman tree

Continue from previous slide:

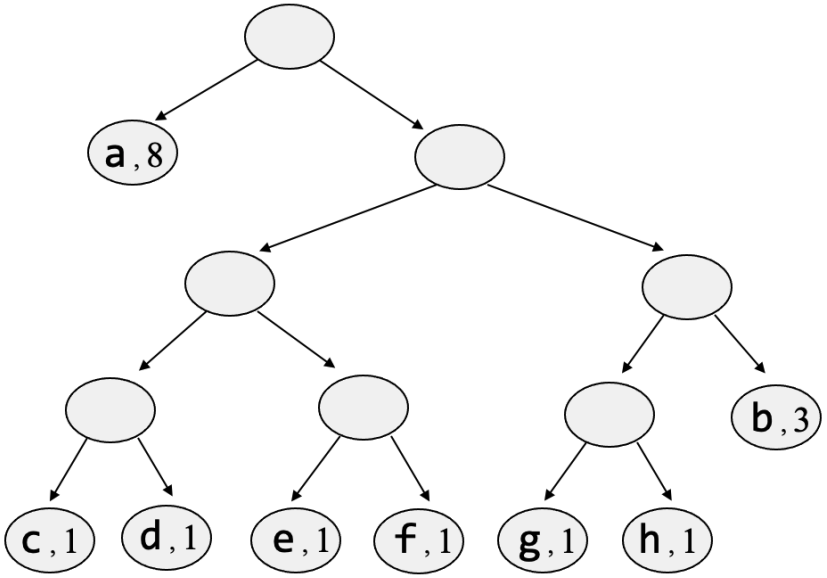


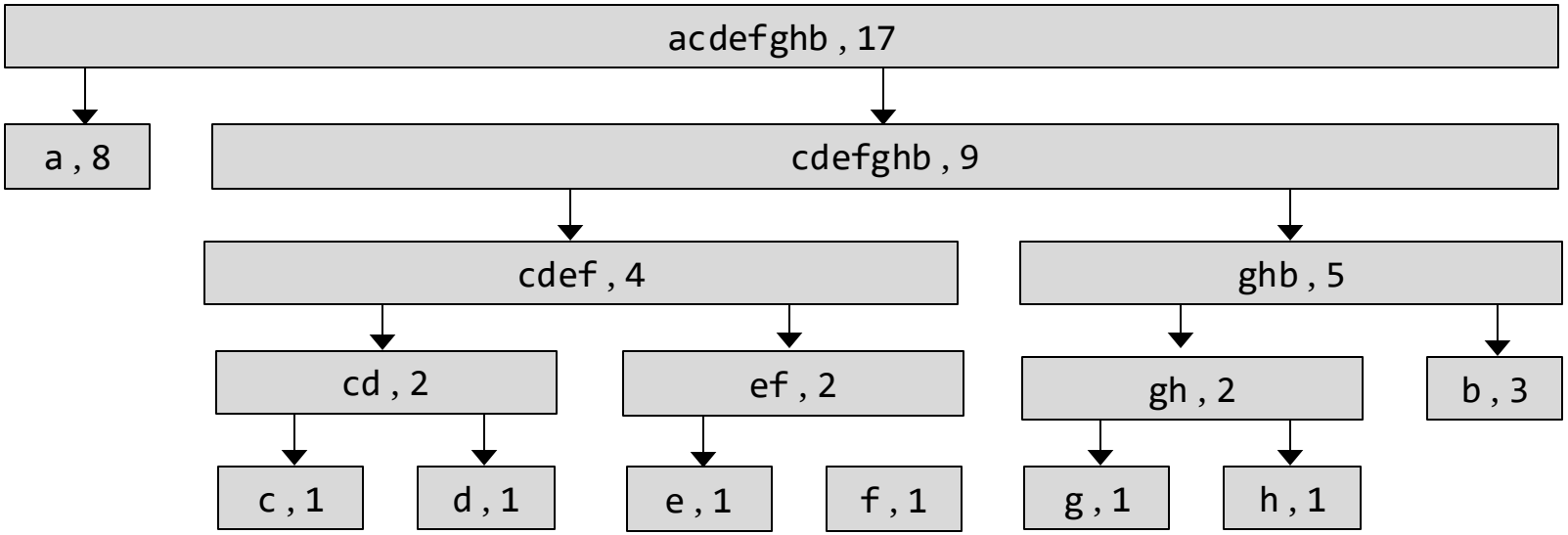
Again:



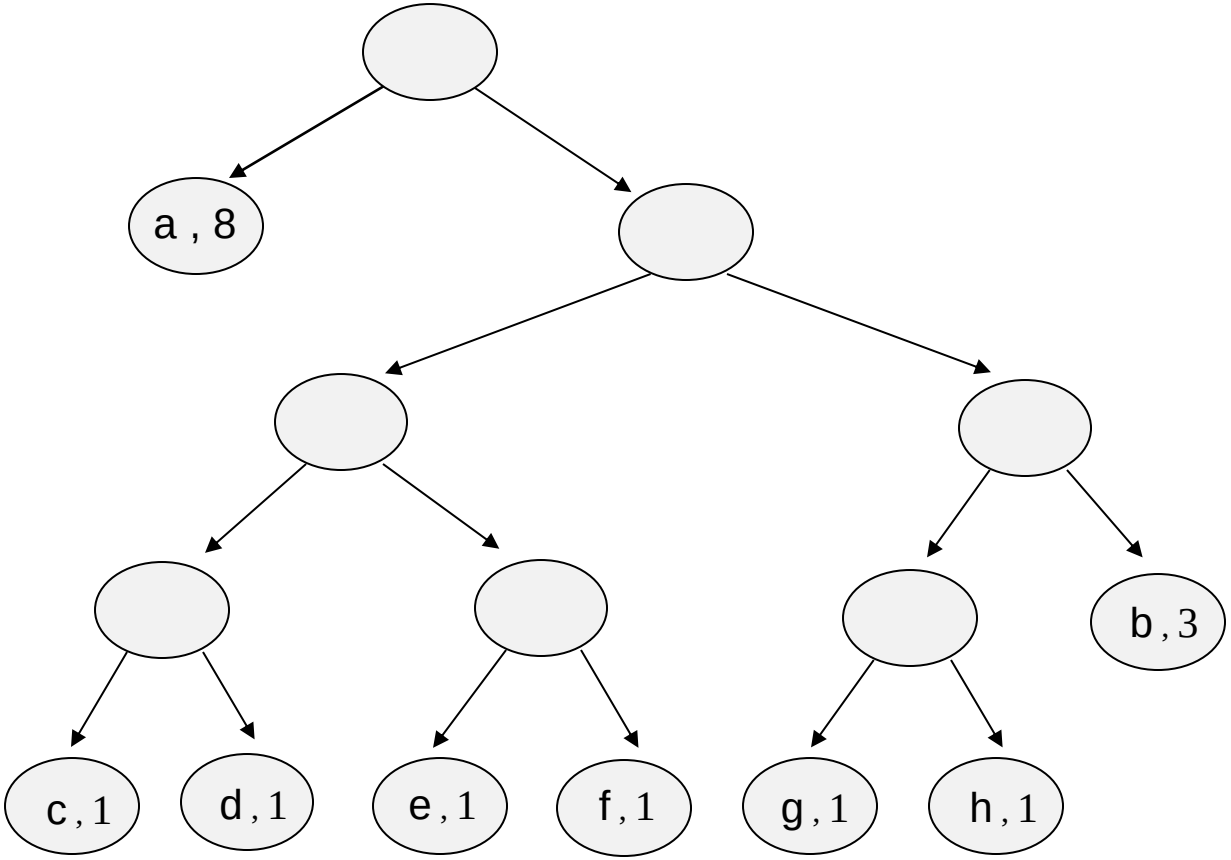
Building a Huffman tree

Binary Tree:

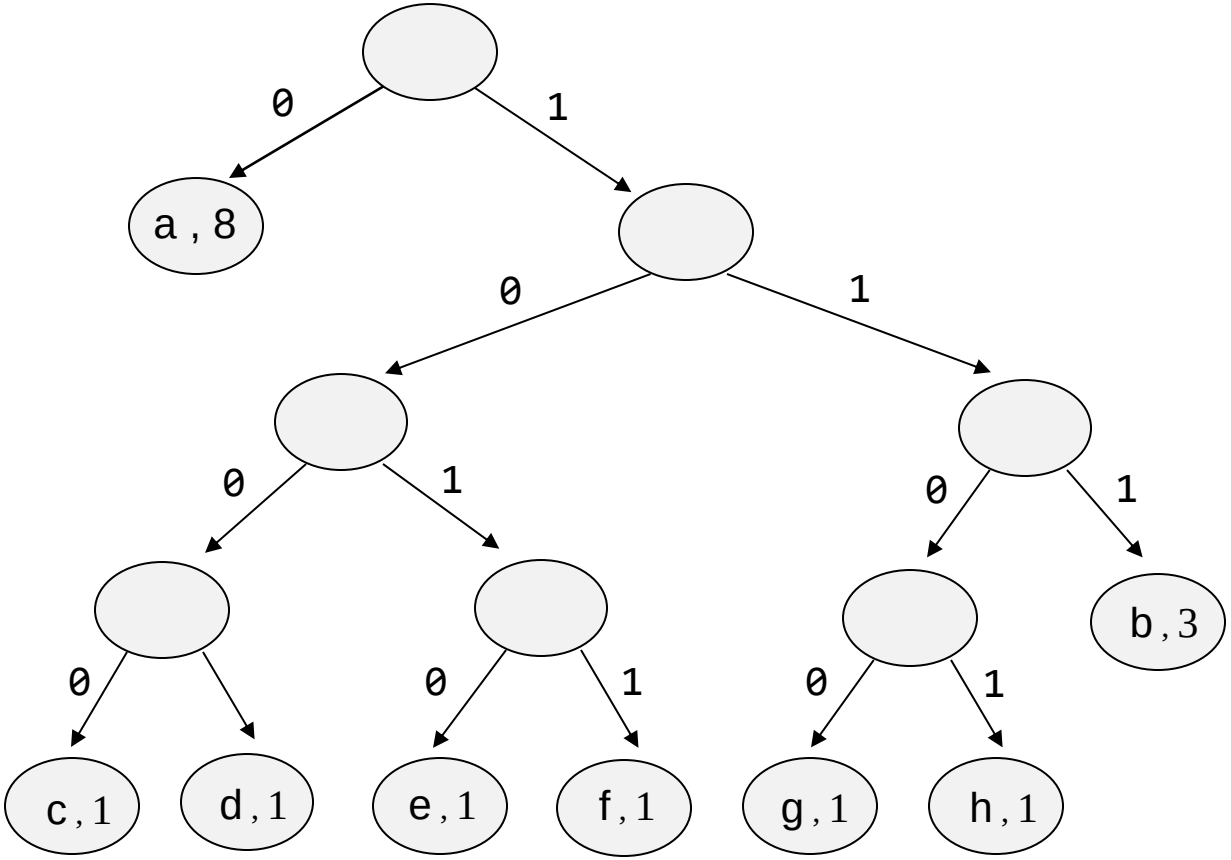




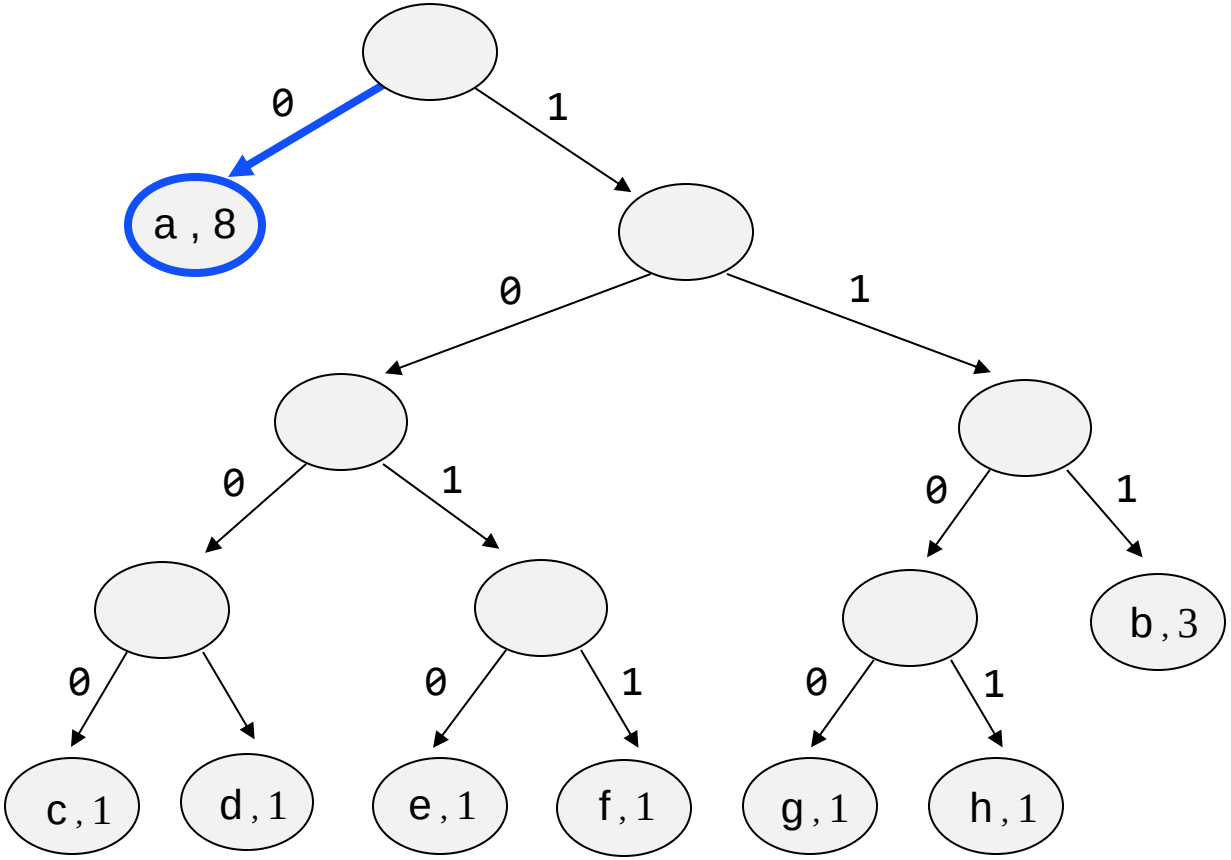
Huffman codes



Huffman codes

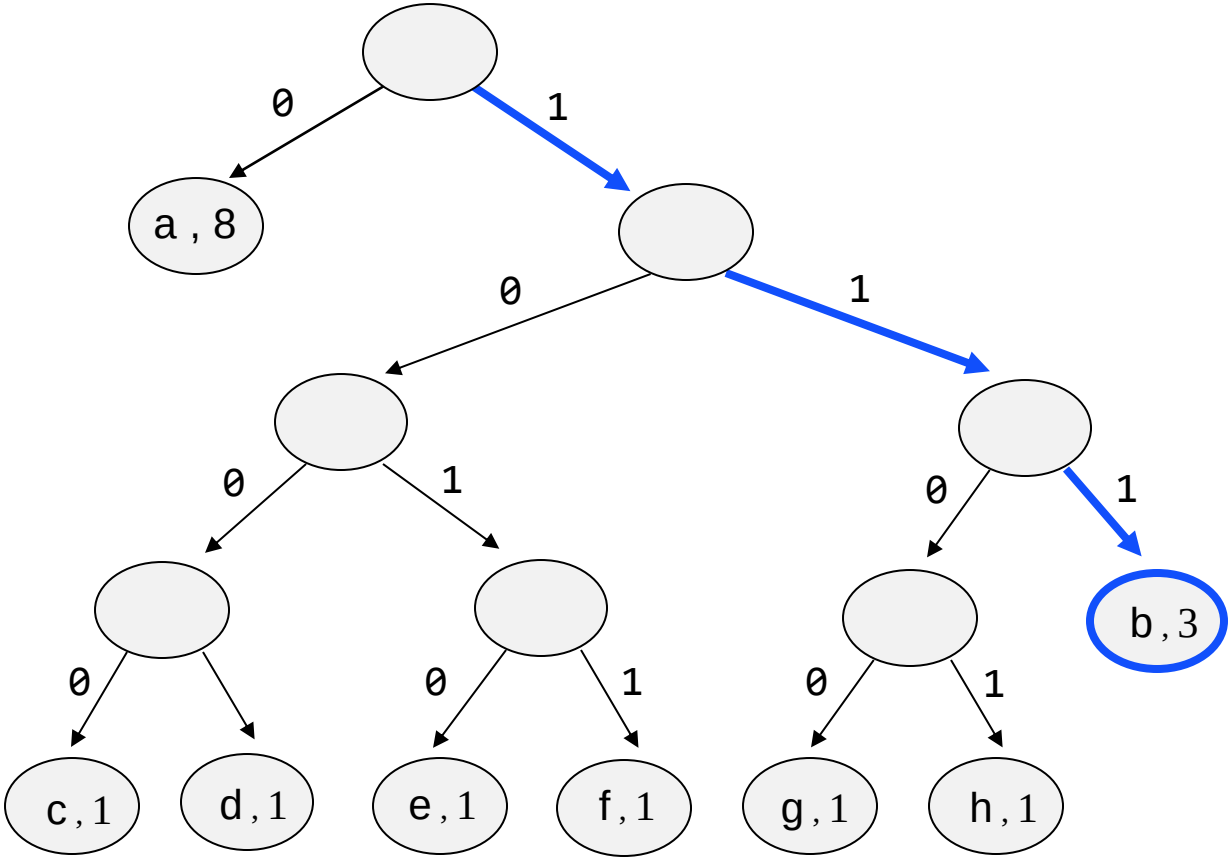


Huffman codes



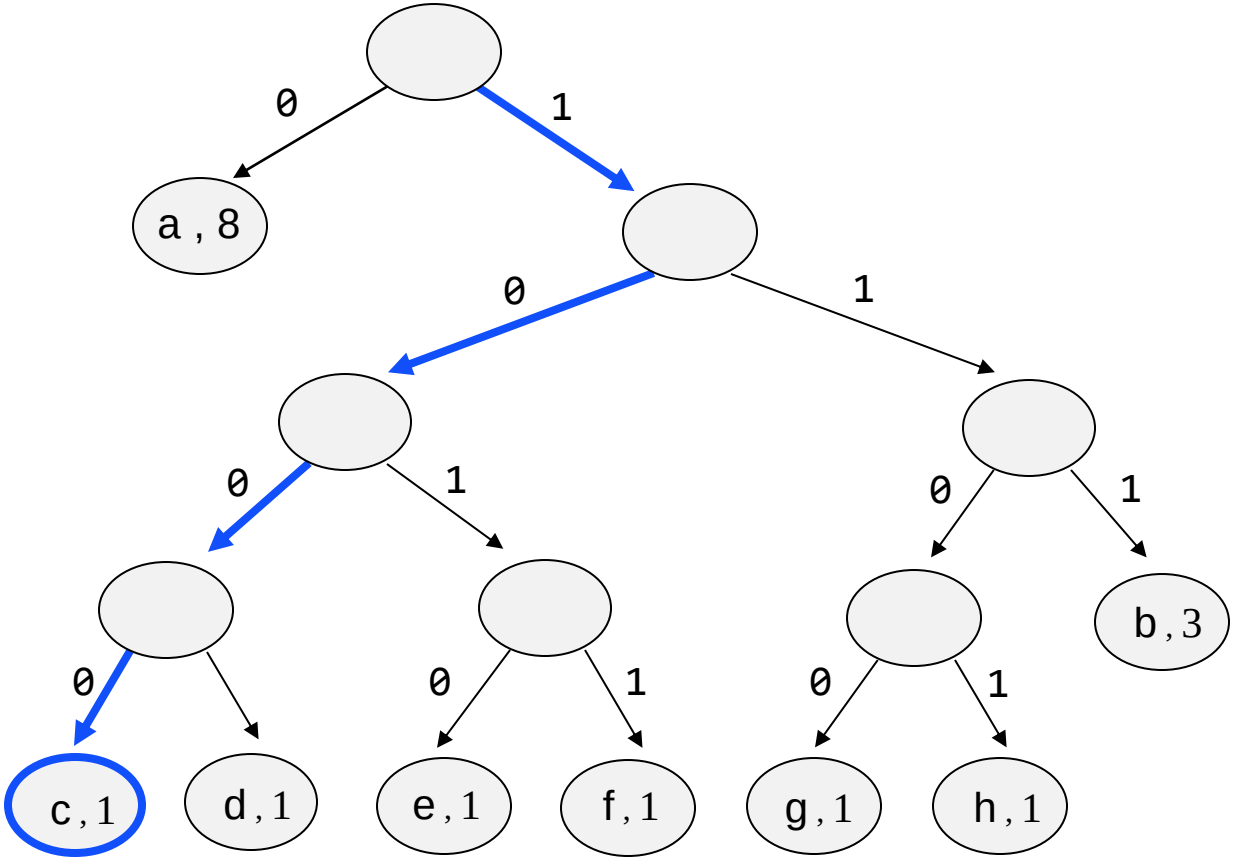

```
code('a') = 0
```

Huffman codes



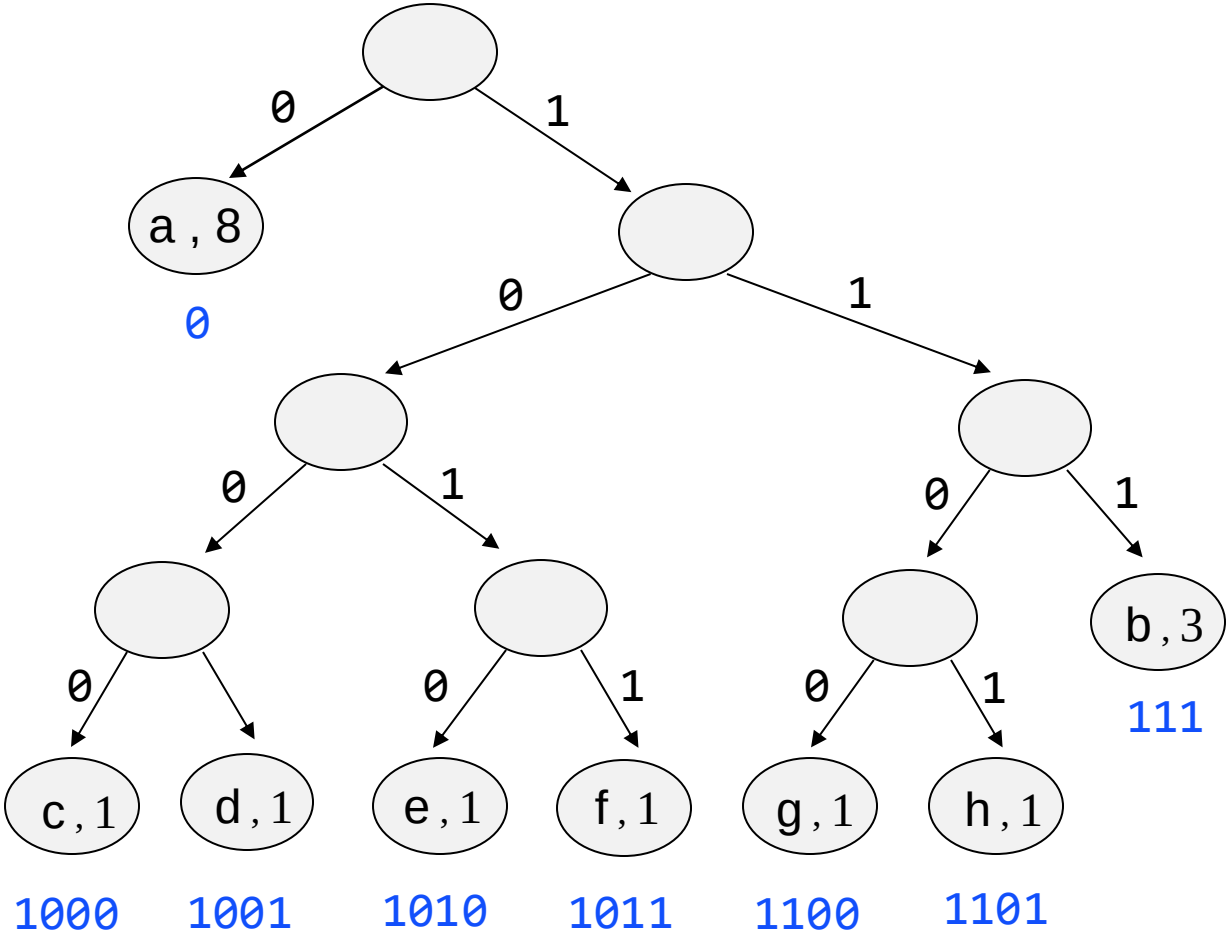
code('b') = 111

Huffman codes

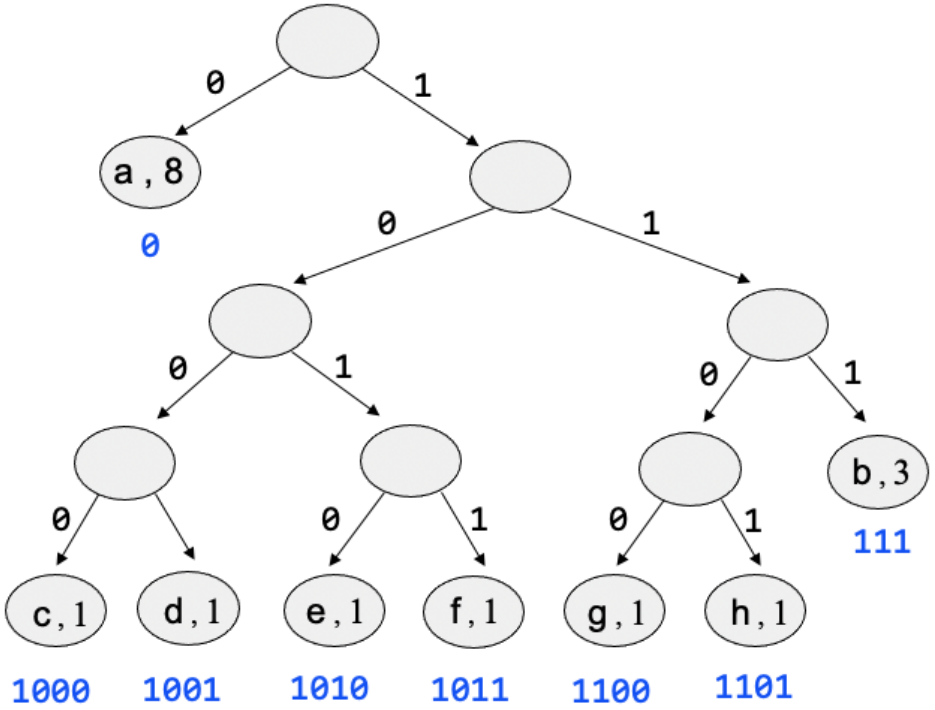


`code('c') = 1000`

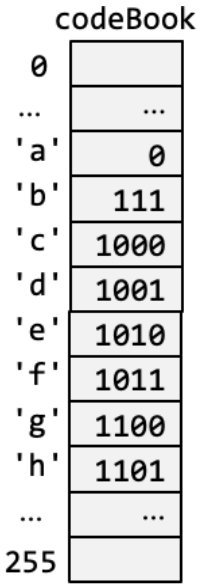
Huffman codes



Huffman codes



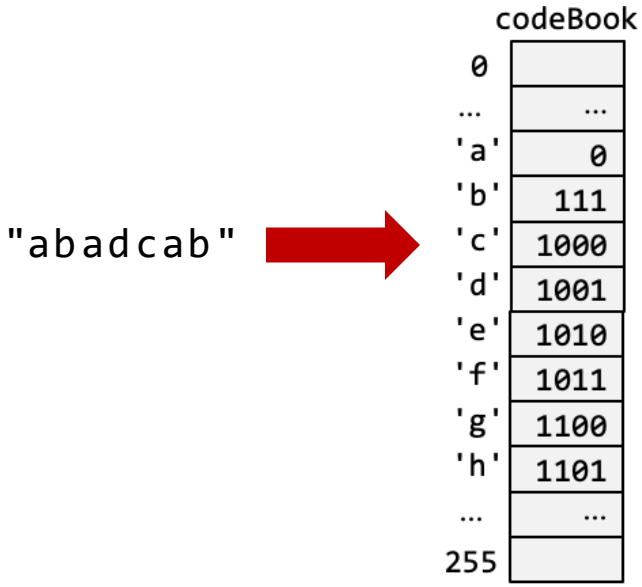
build



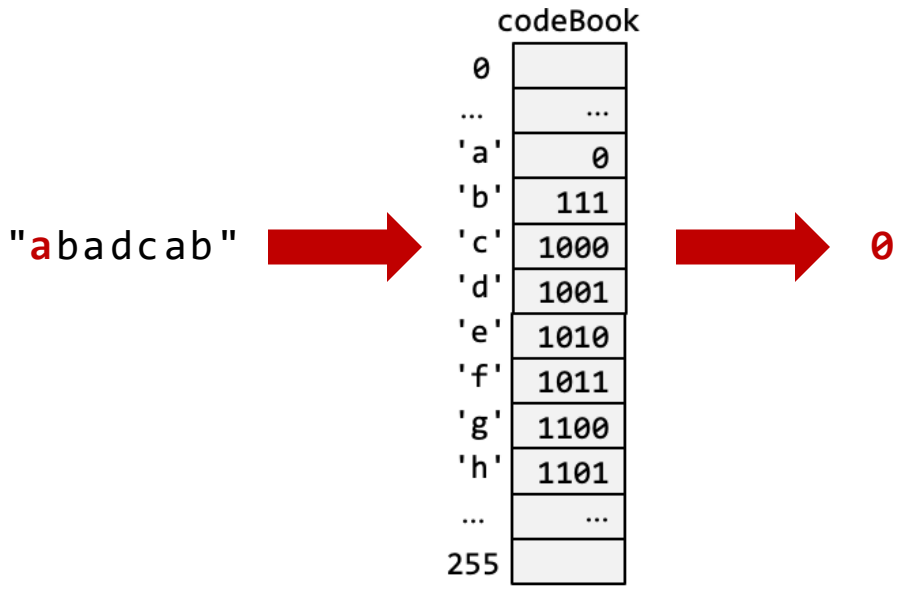
Coding properties

- **Efficiency:** The higher the character frequency, the shorter the code
- **Prefix free:** No code is a prefix of any other code

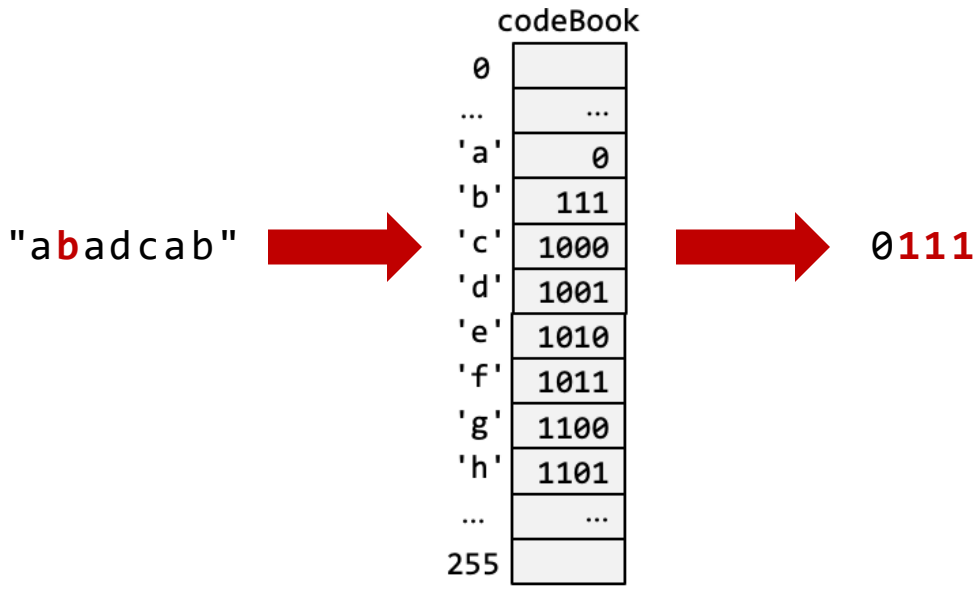
Compression



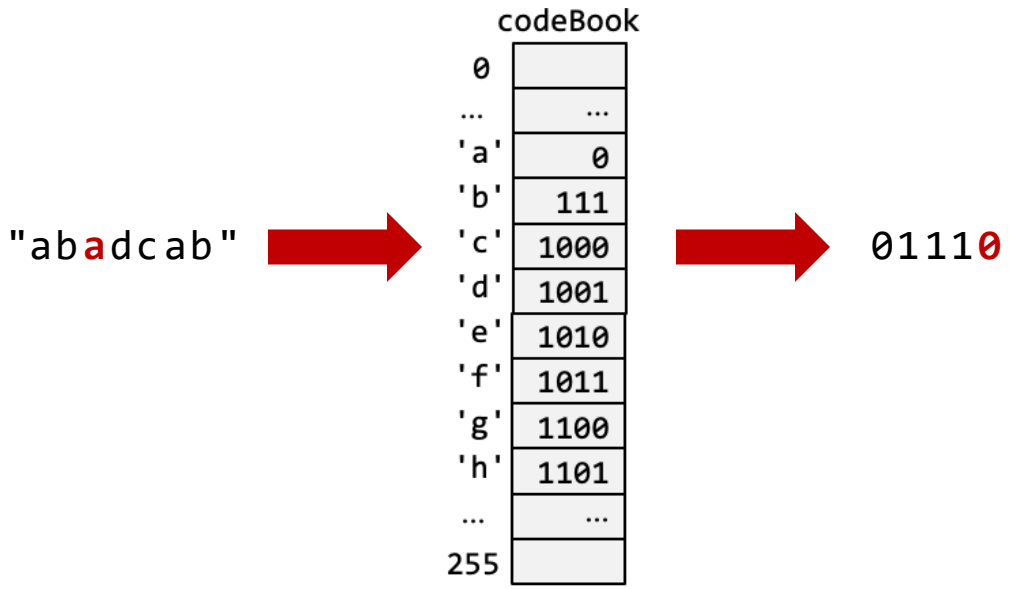
Compression



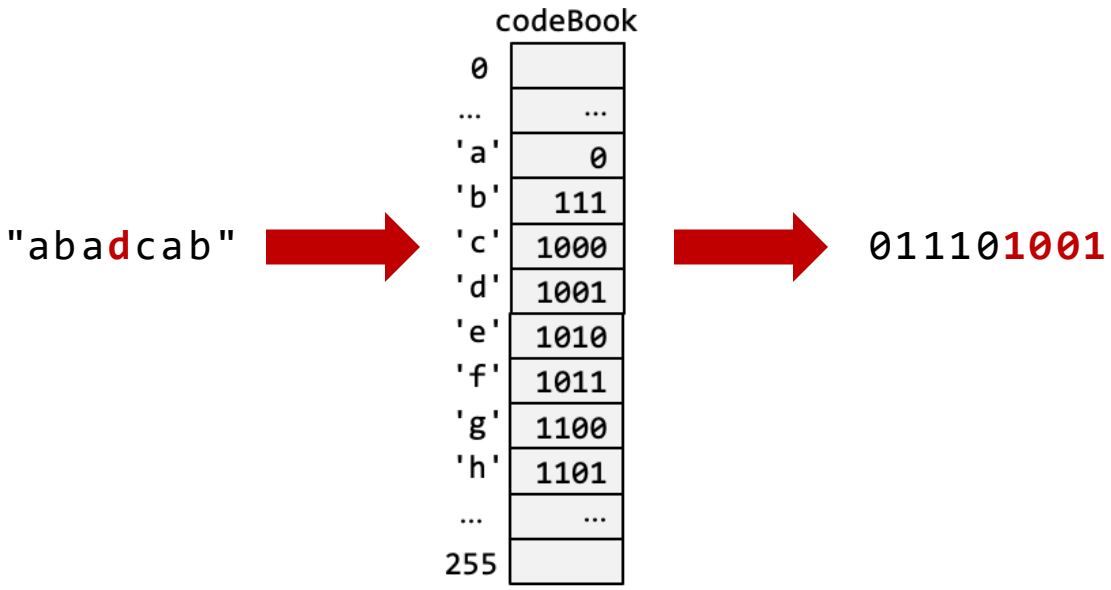
Compression



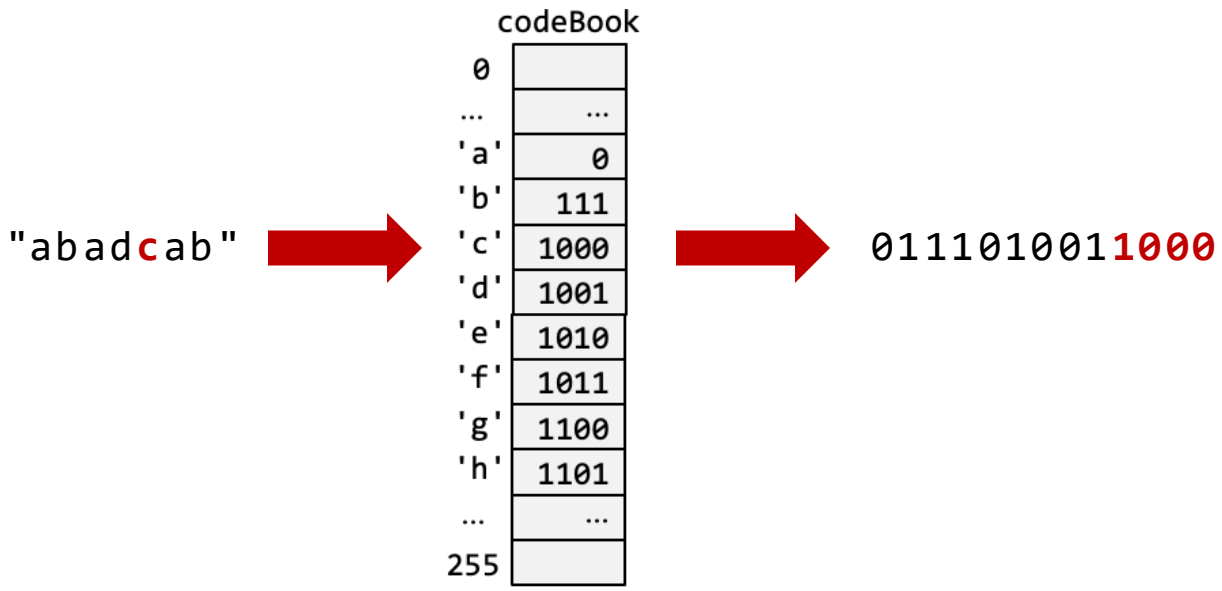
Compression



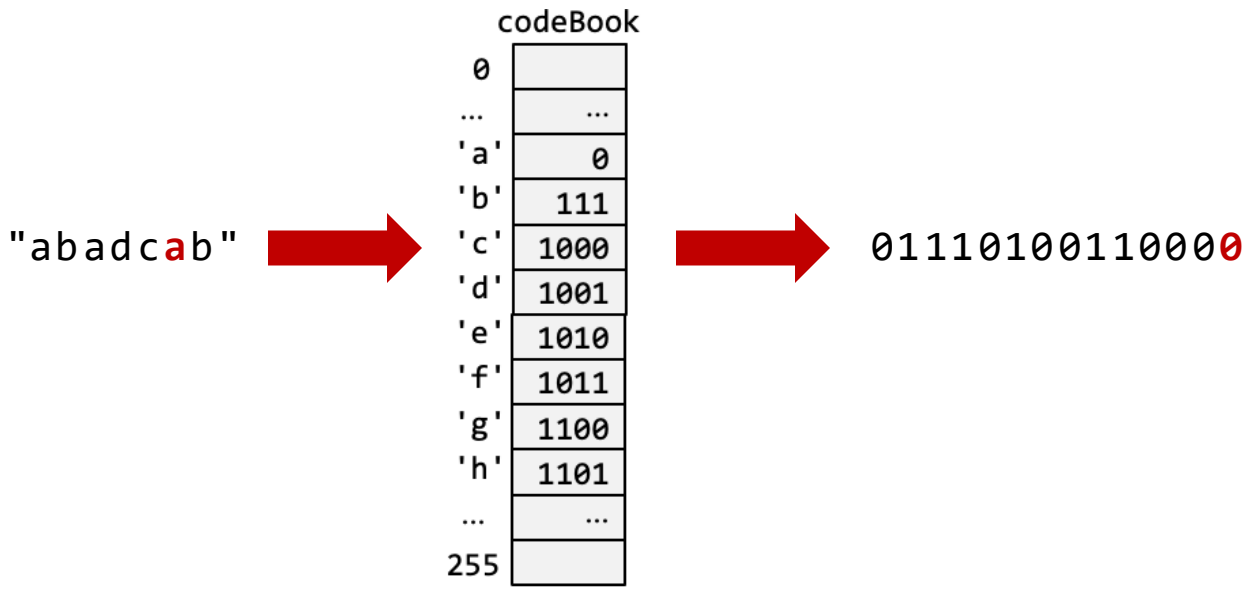
Compression



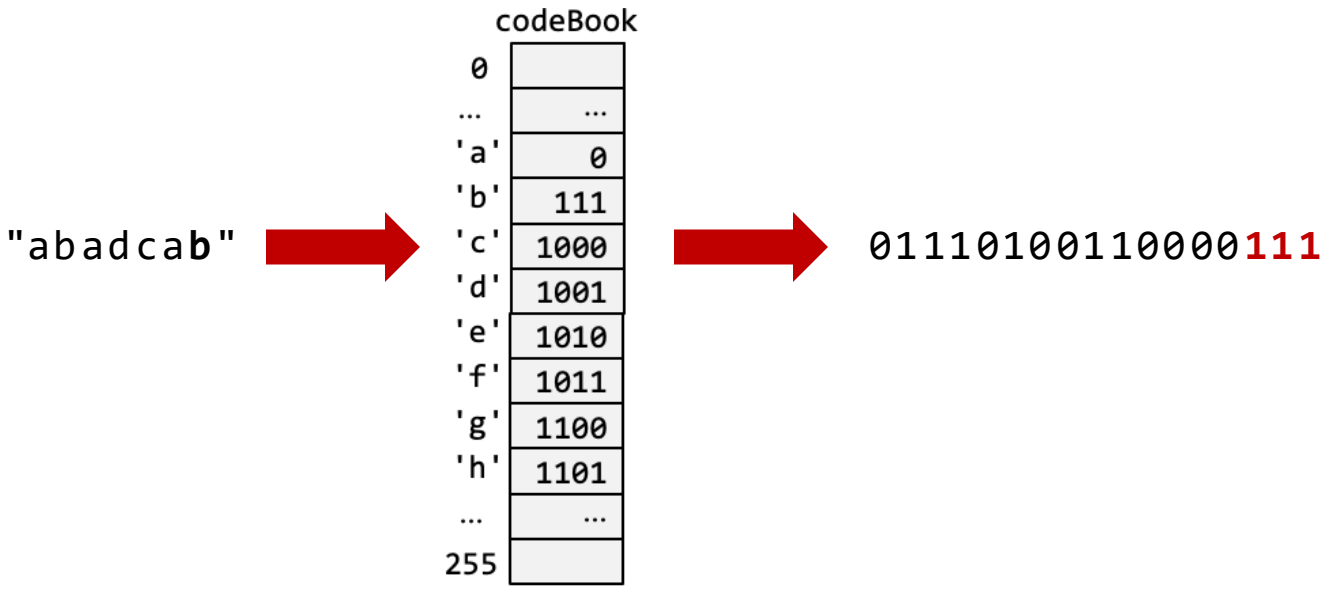
Compression



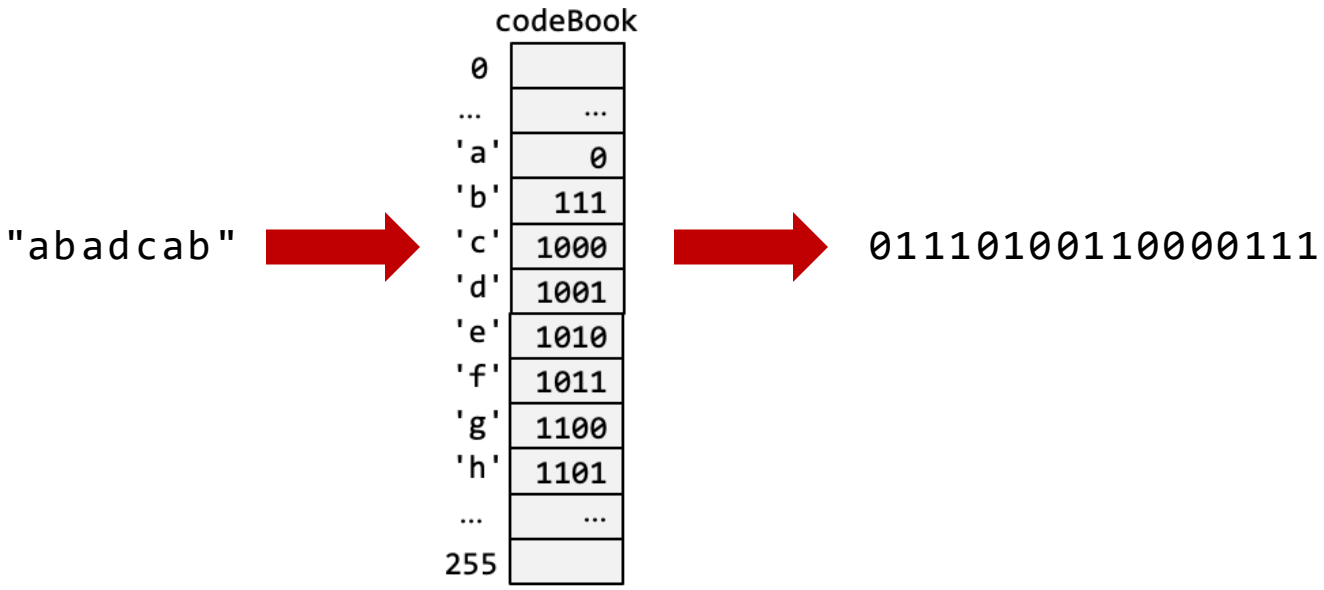
Compression



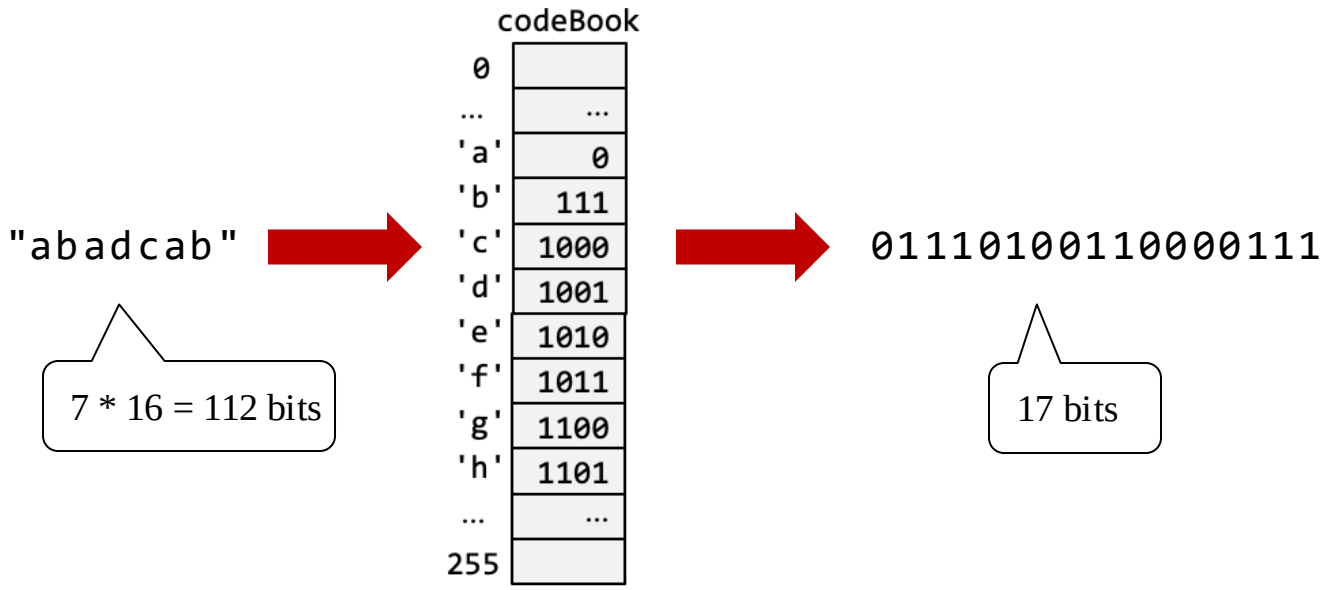
Compression



Compression



Compression

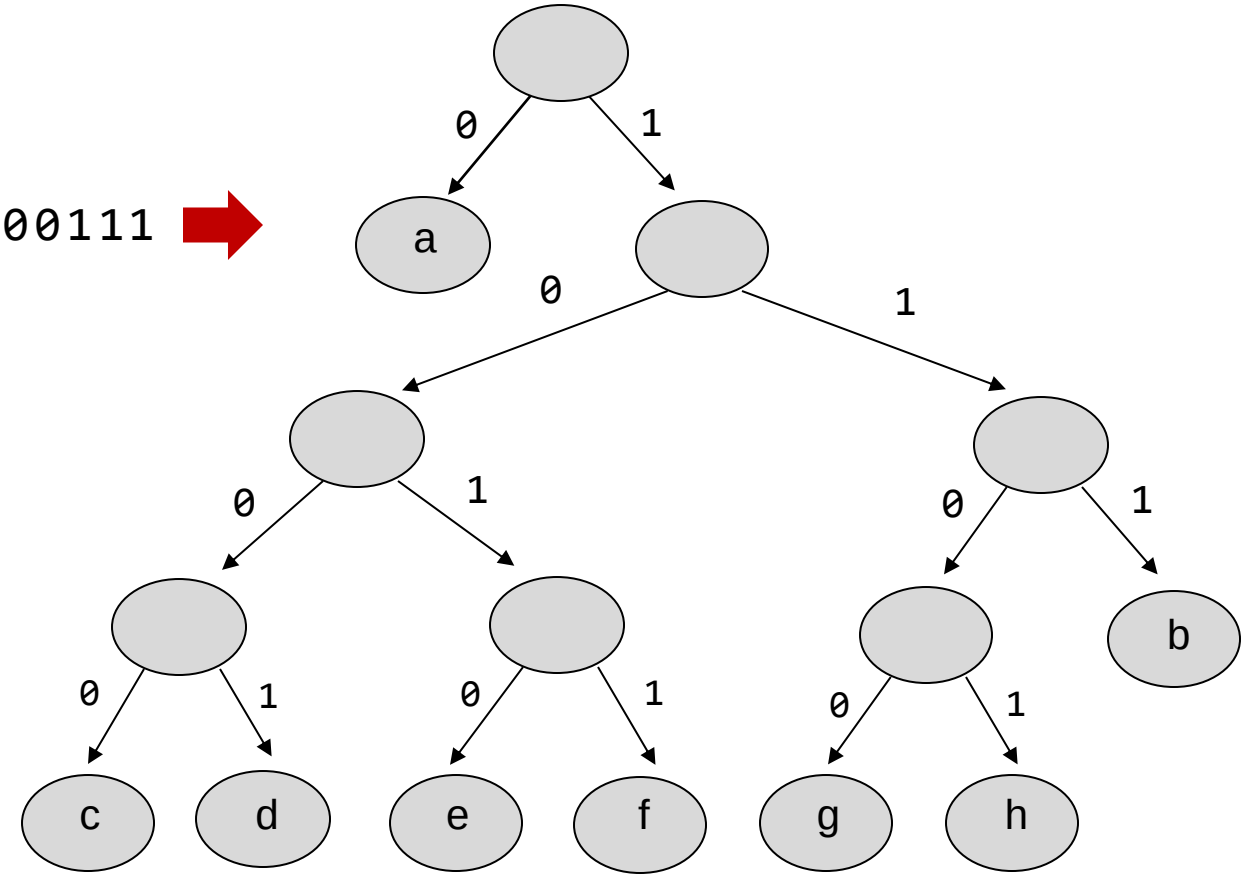


Compressed text: 84% shorter

Decompression

Compressed text:

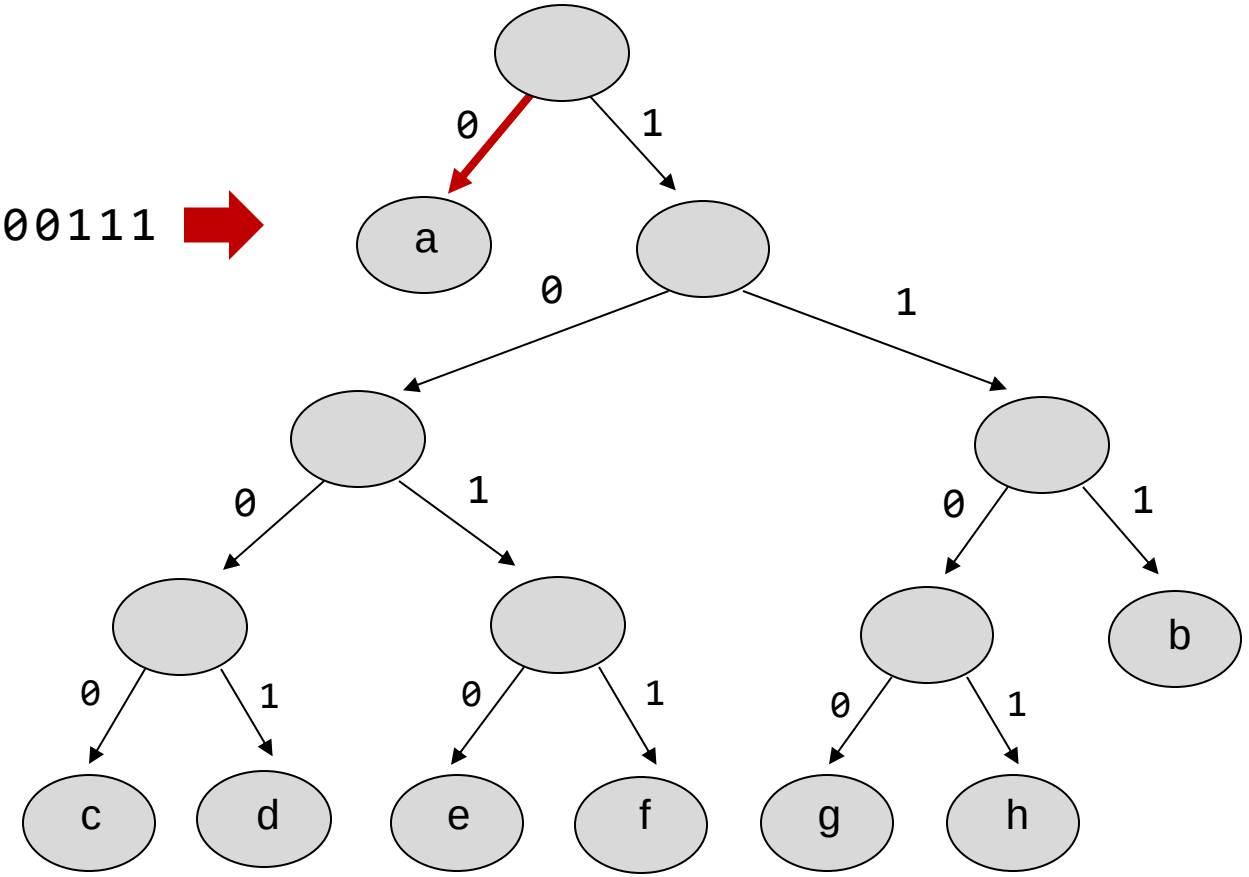
`01110100110000111`



Decompression

Compressed text:

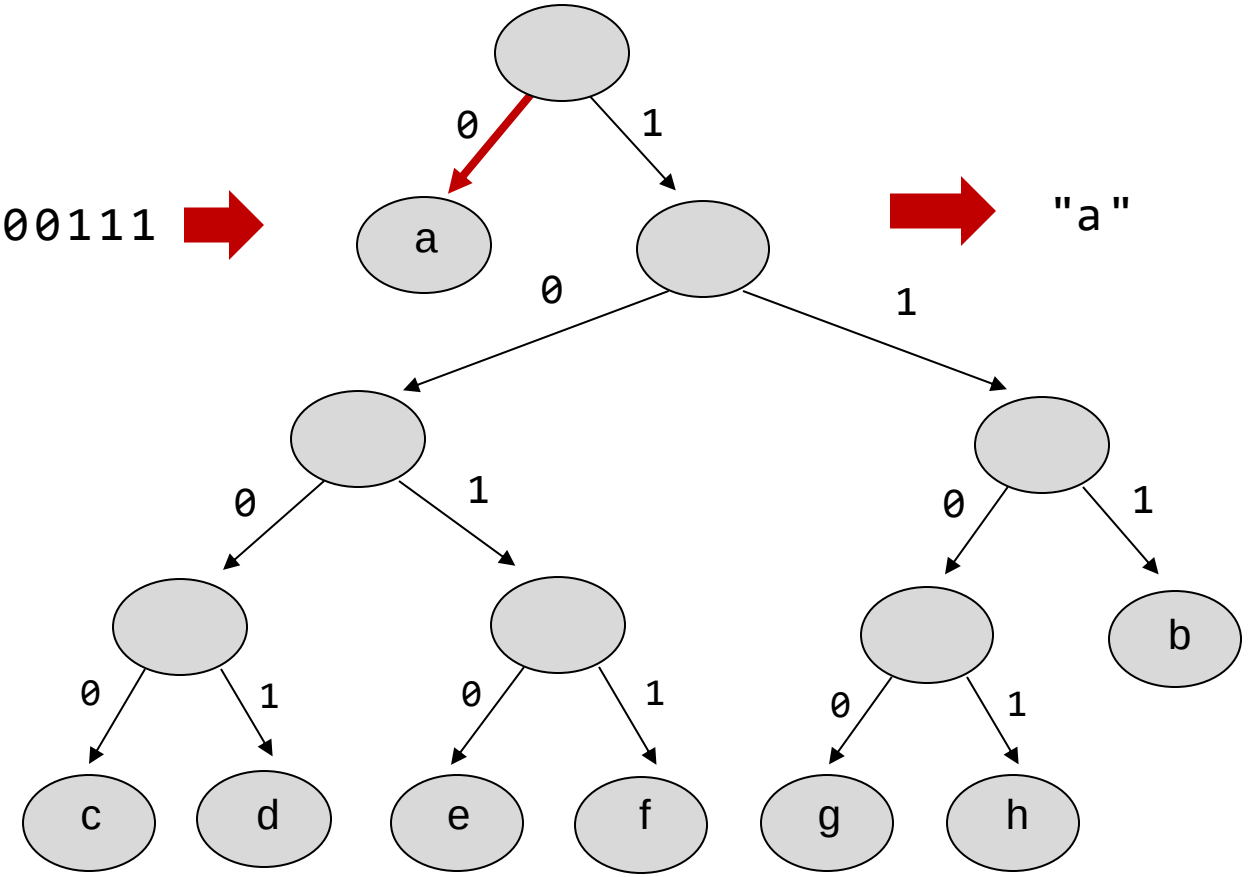
0111010011000111



Decompression

Compressed text:

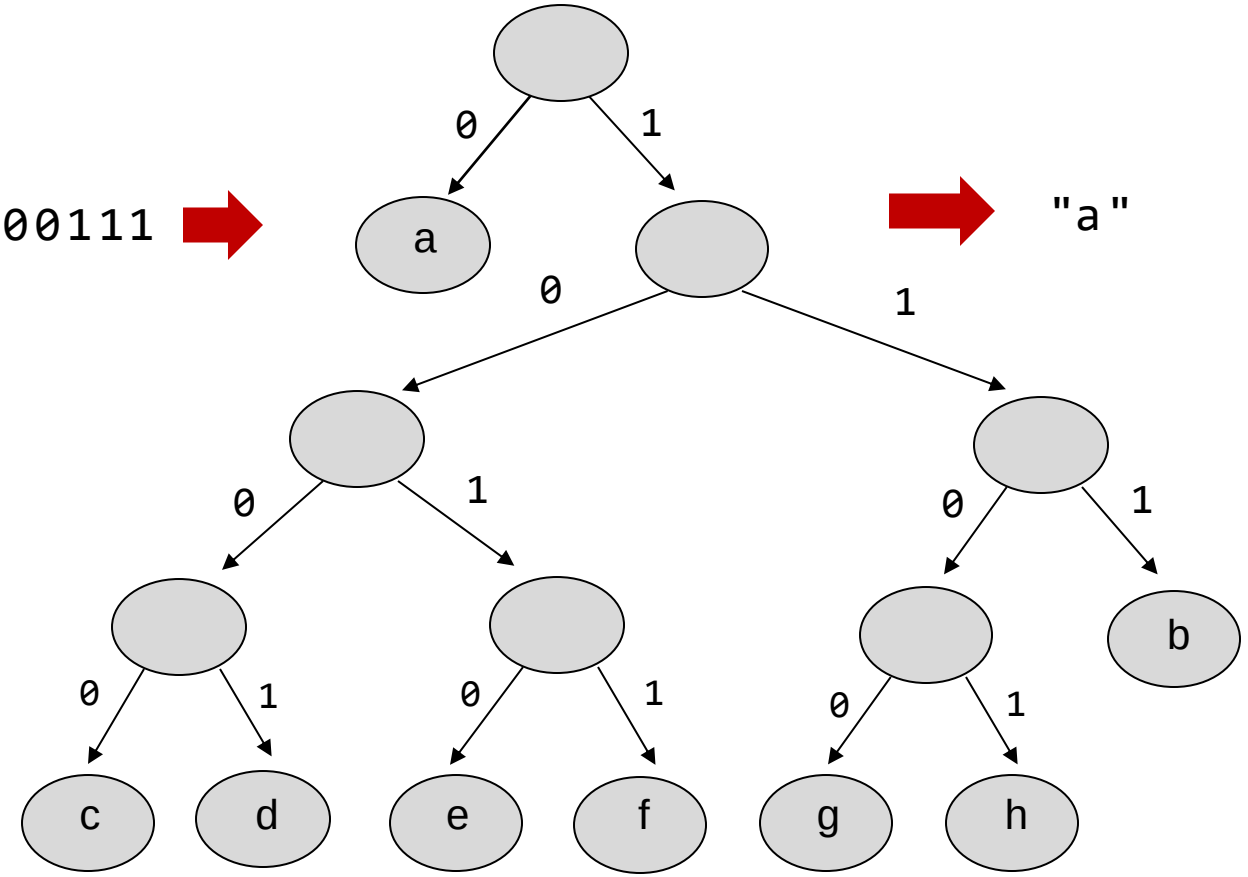
0111010011000111



Decompression

Compressed text:

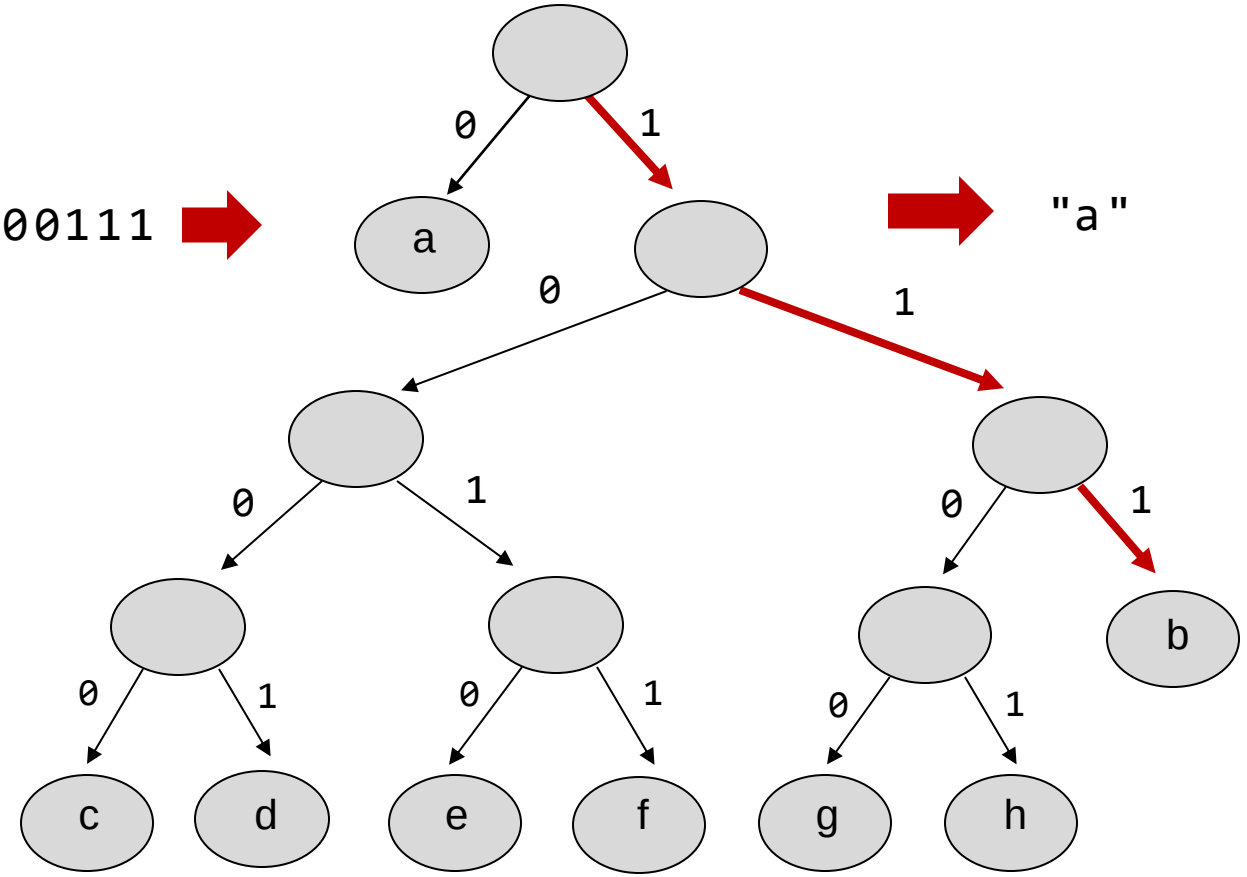
```
01110100110000111
```



Decompression

Compressed text:

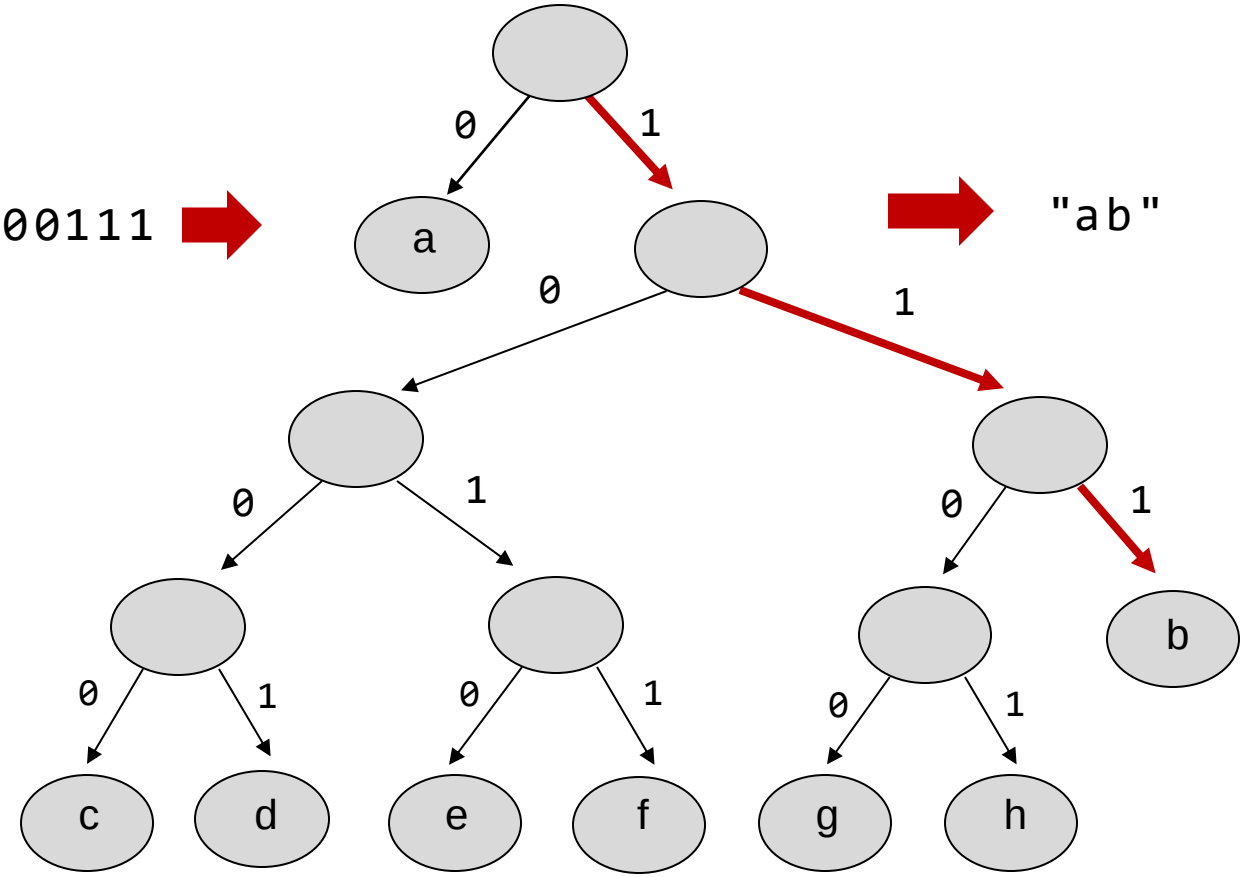
0111010011000111



Decompression

Compressed text:

0111010011000111

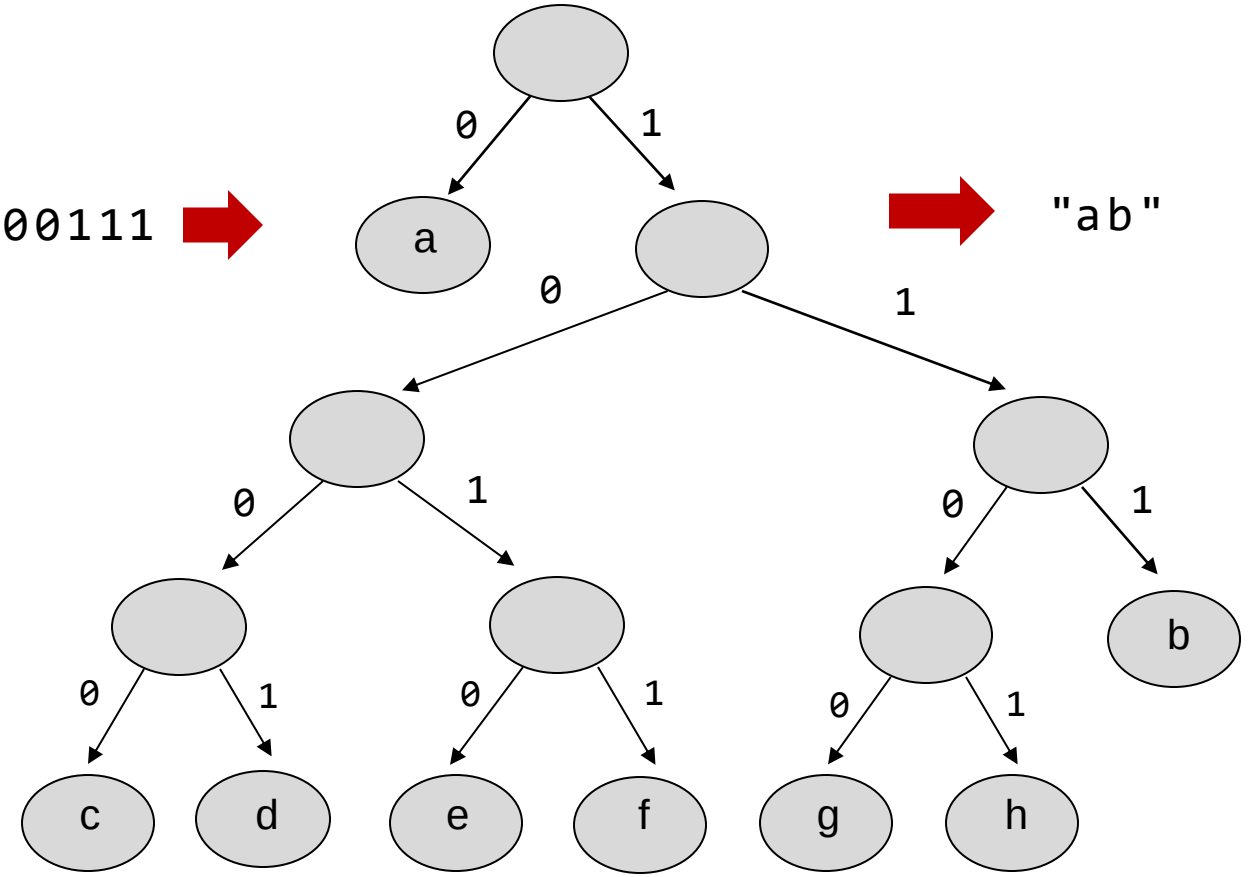


"ab"

Decompression

Compressed text:

```
01110100110000111
```

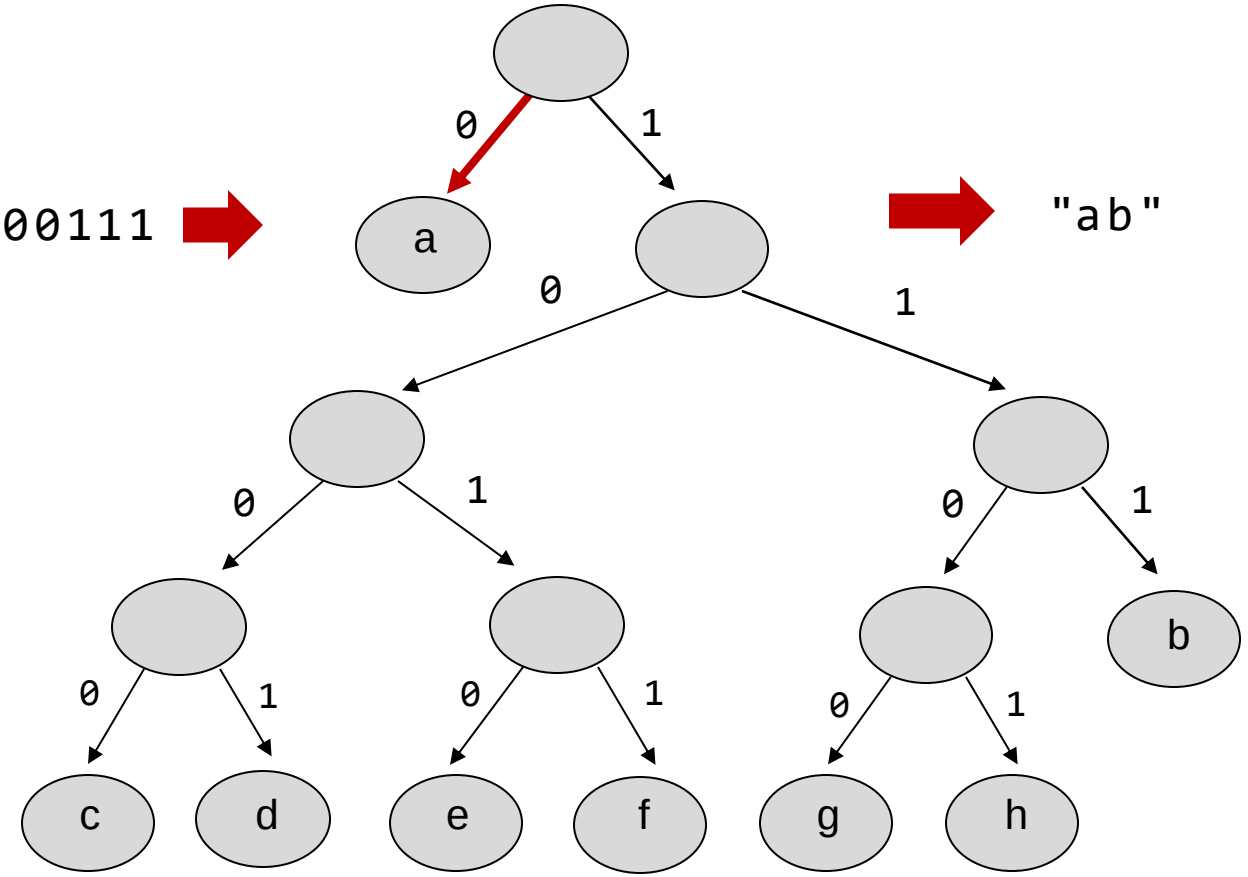


"ab"

Decompression

Compressed text:

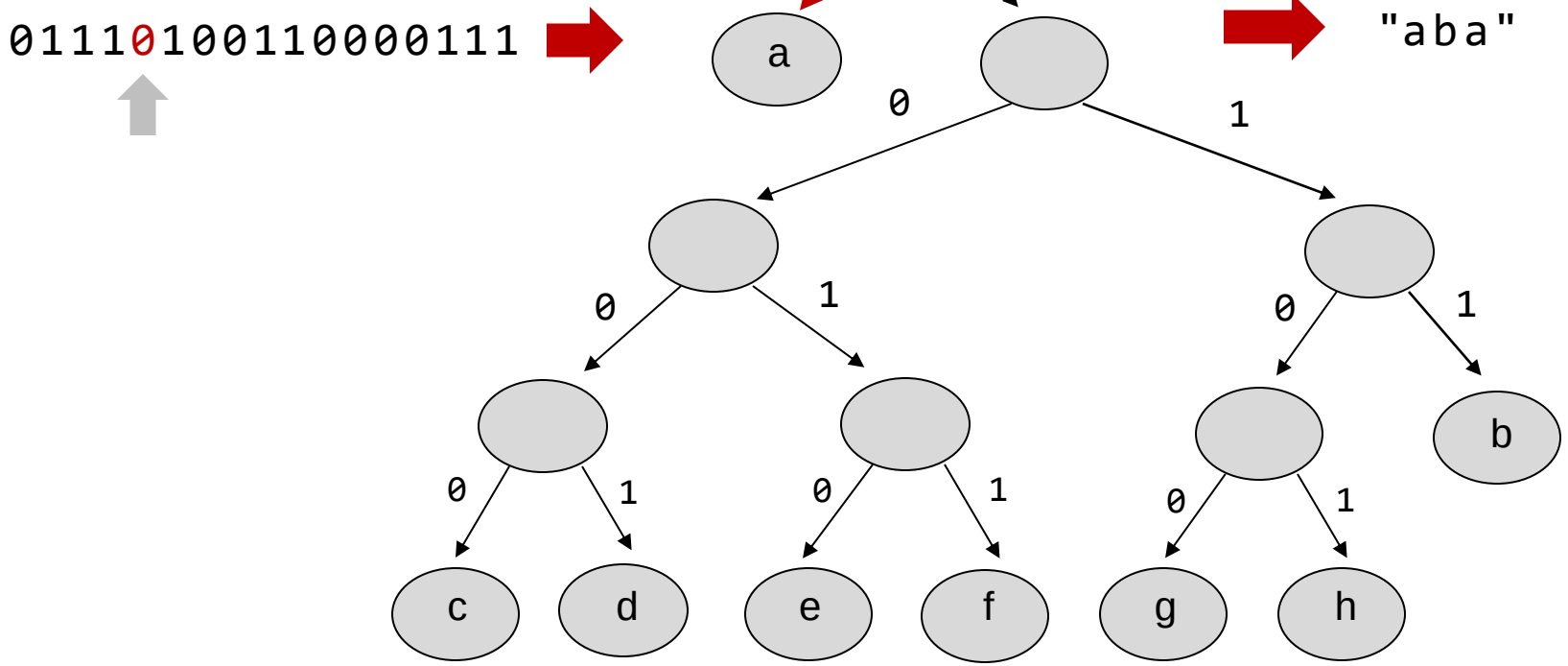
0111**0**100110000111



"ab"

Decompression

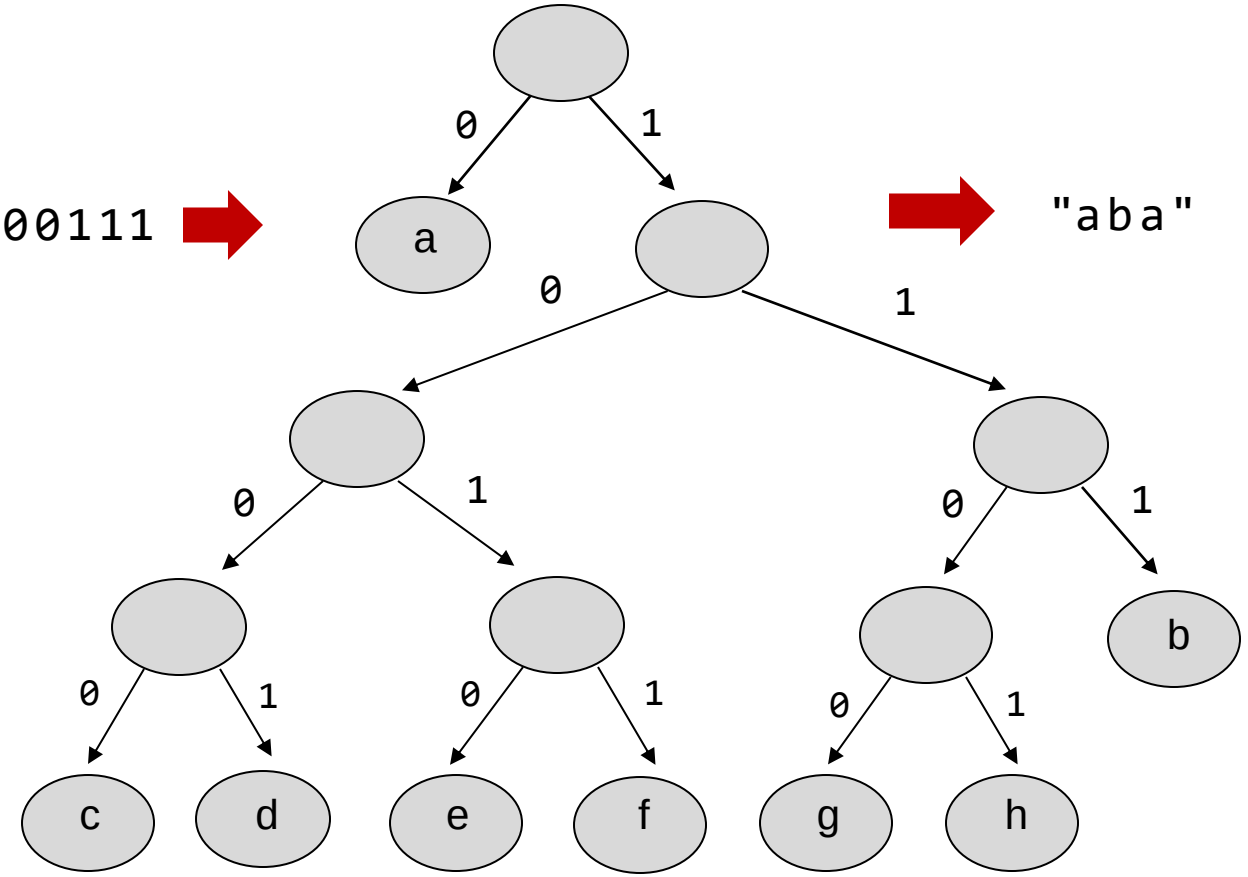
Compressed text:



Decompression

Compressed text:

`01110100110000111`

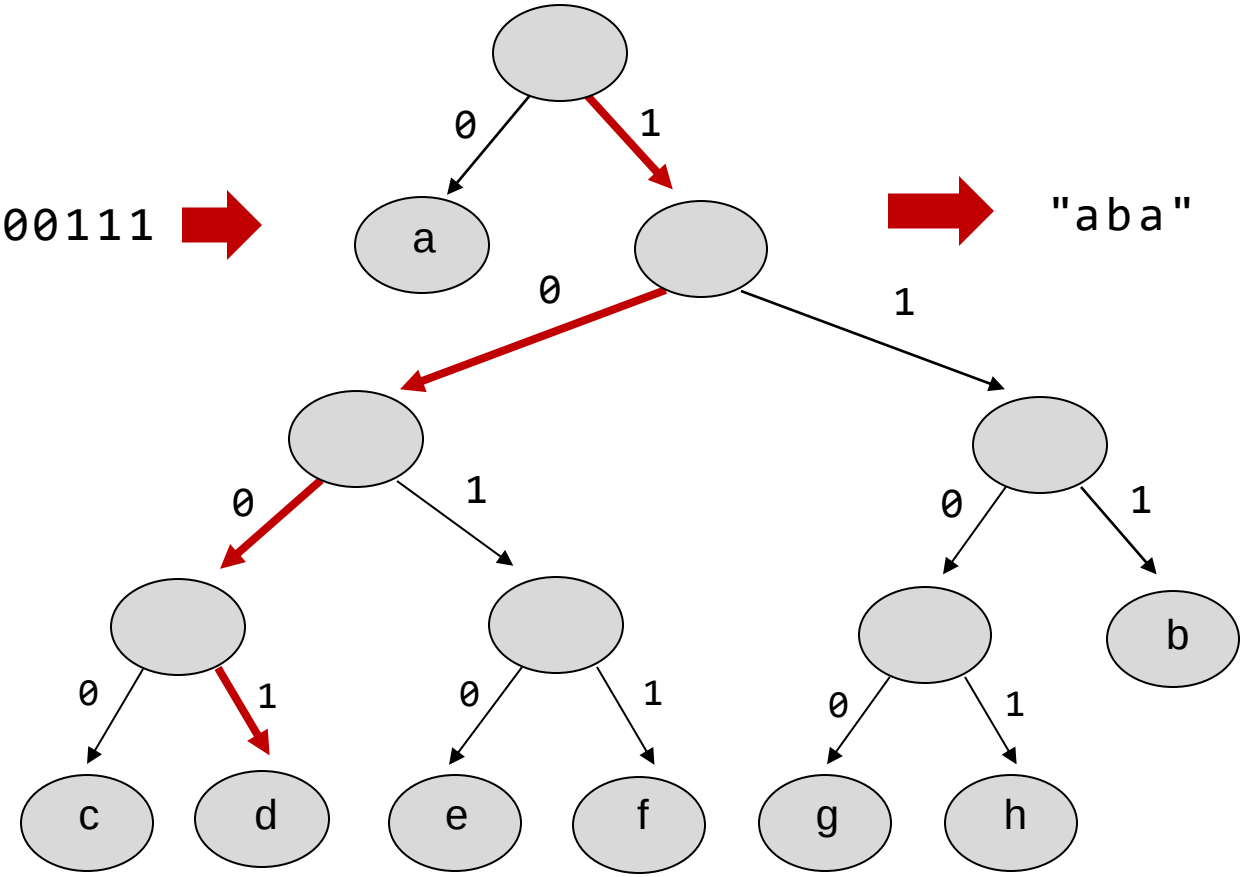


"aba"

Decompression

Compressed text:

01110100110000111

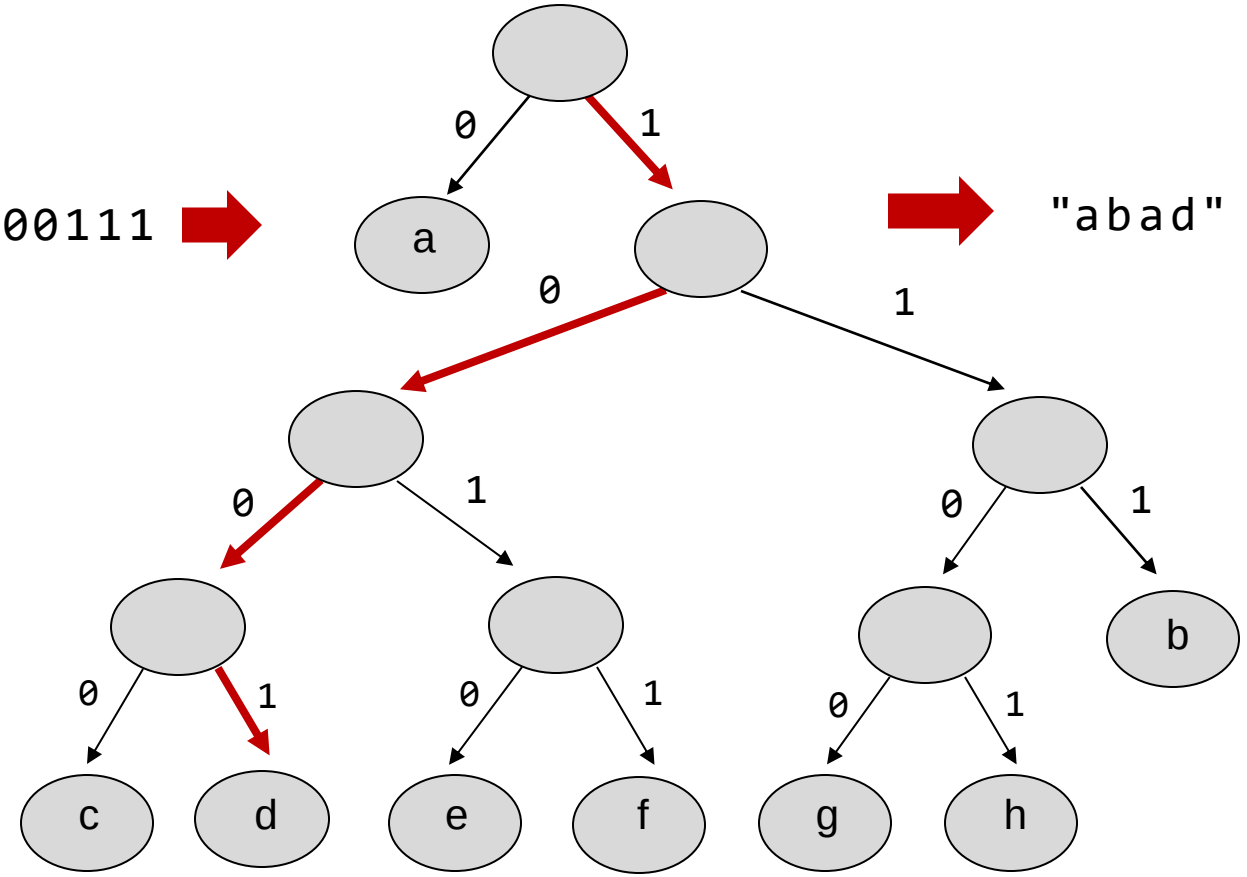


"aba"

Decompression

Compressed text:

01110100110000111

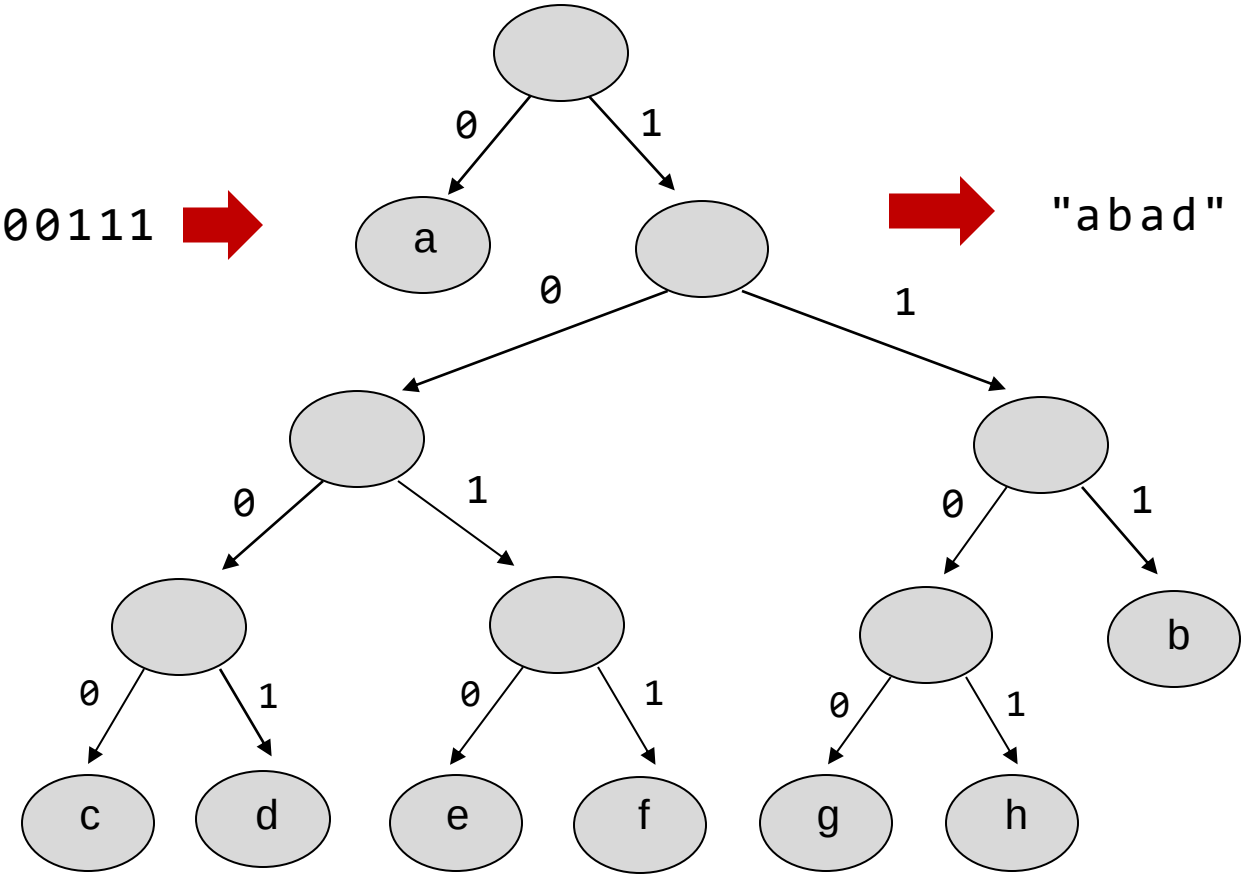


"abad"

Decompression

Compressed text:

```
01110100110000111
```

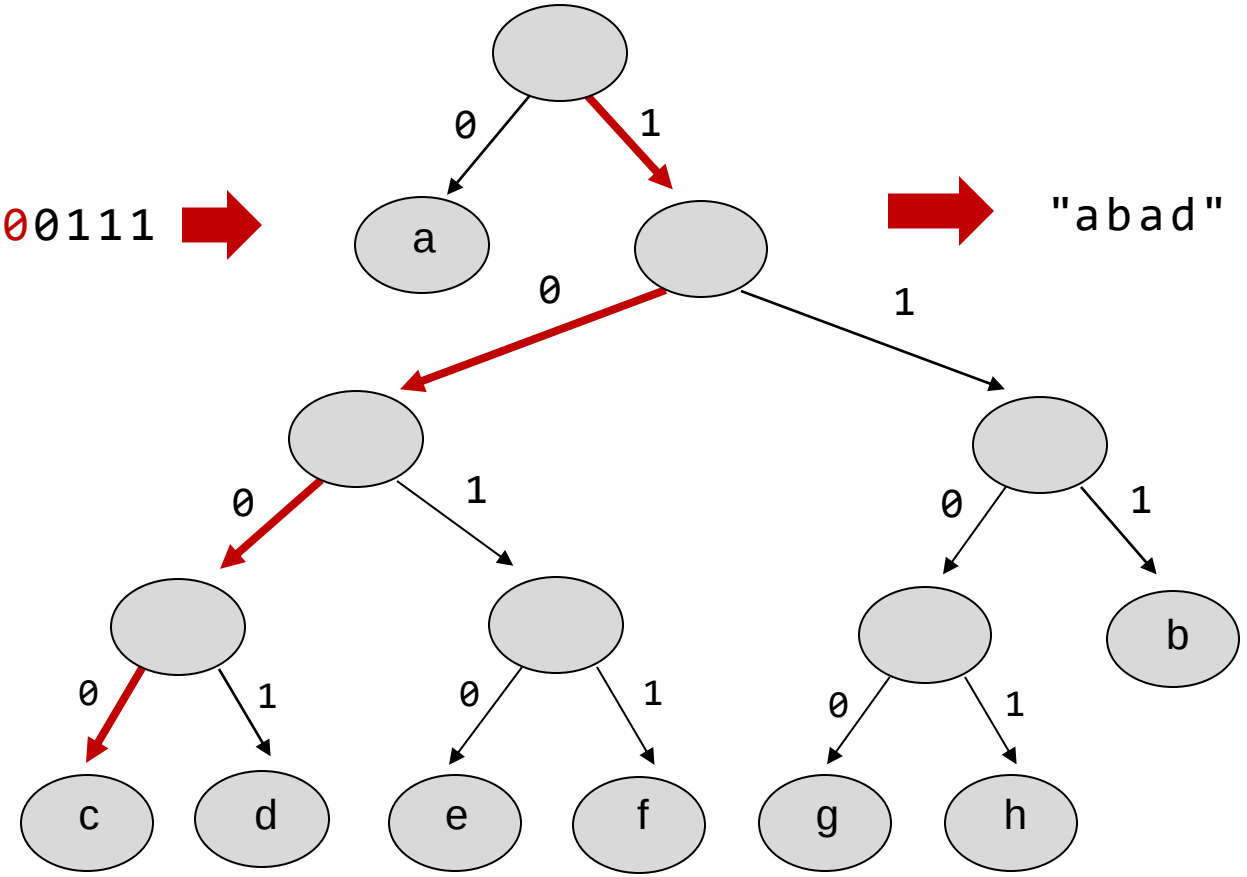


"abad"

Decompression

Compressed text:

01110100110000111

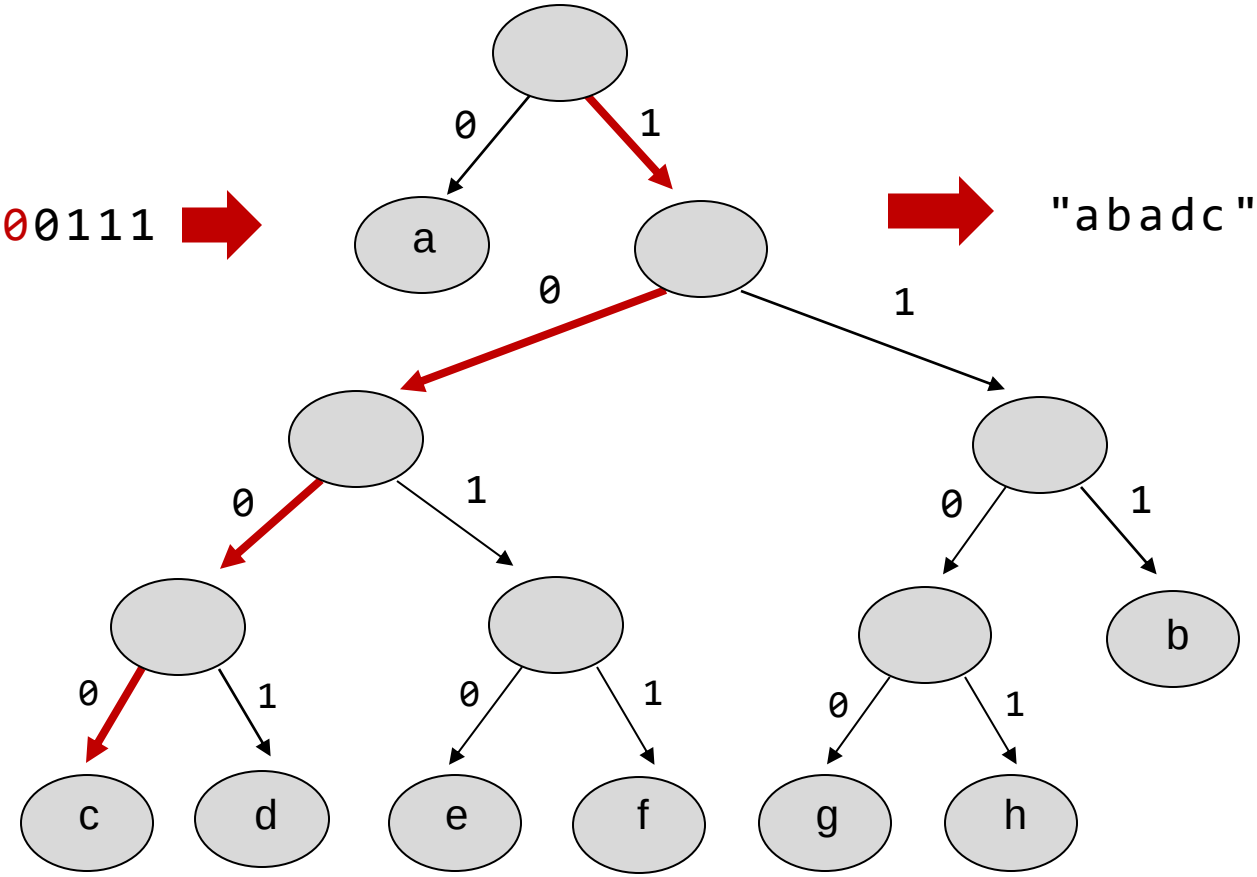


"abad"

Decompression

Compressed text:

01110100110000111

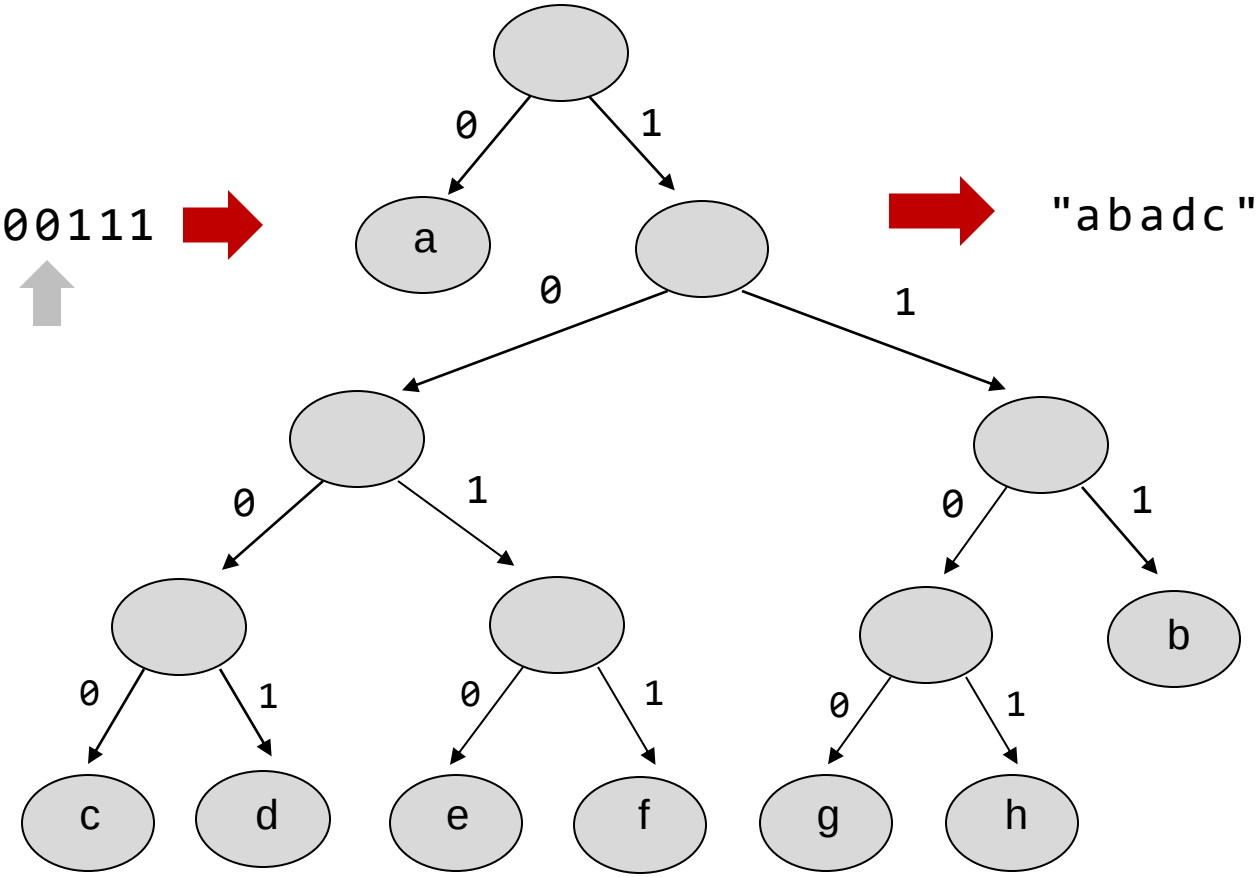


"abadc"

Decompression

Compressed text:

```
01110100110000111
```

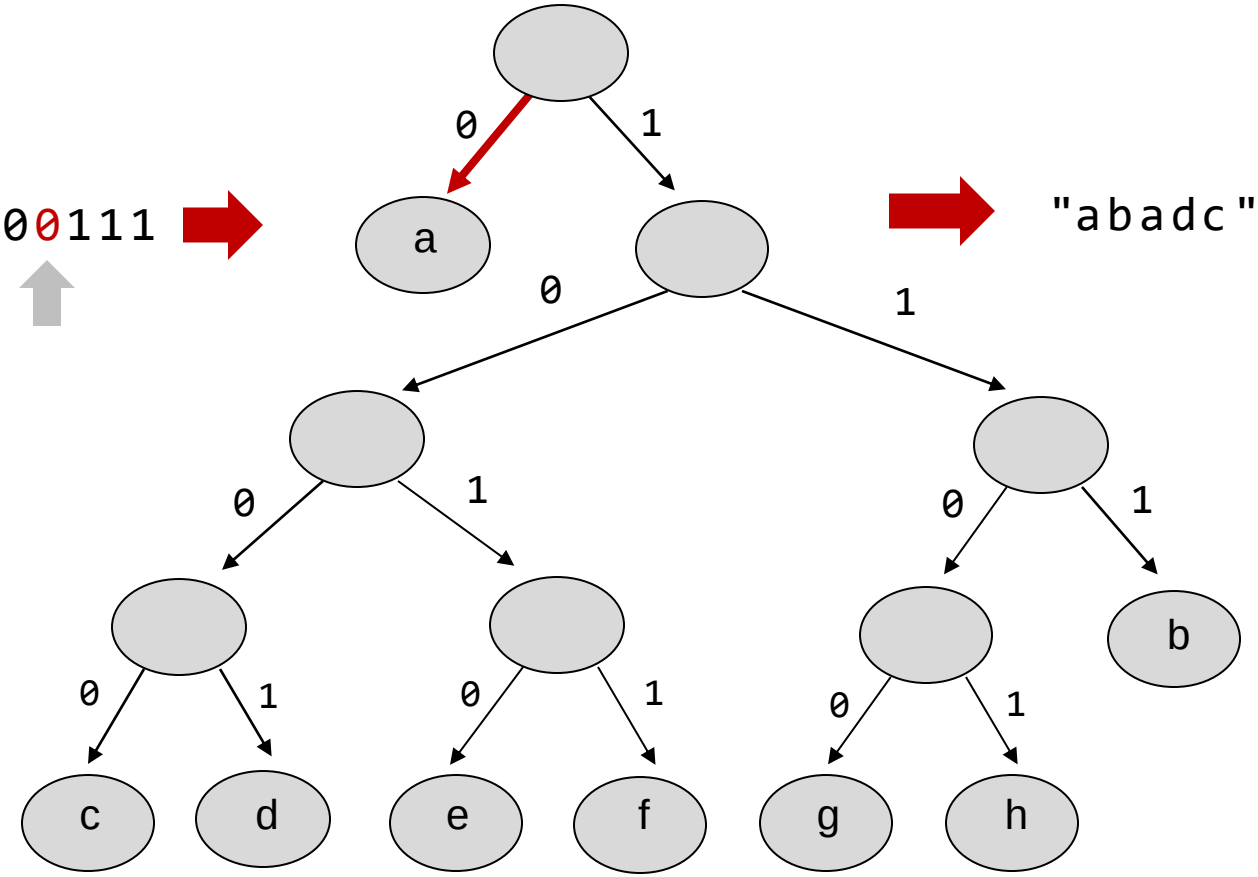


"abadc"

Decompression

Compressed text:

`0111010011000`**`0`**`111`

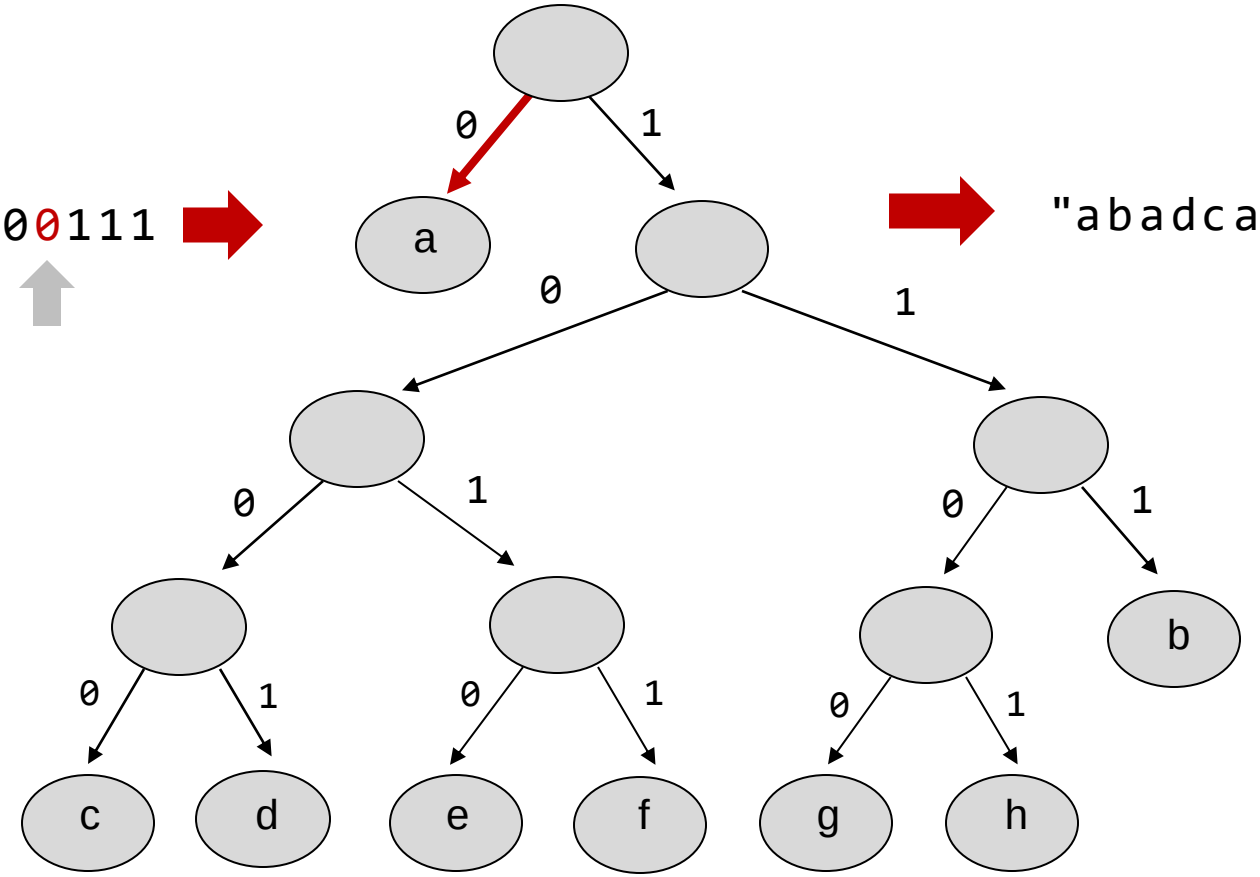


"abadc"

Decompression

Compressed text:

`01110100110000111`

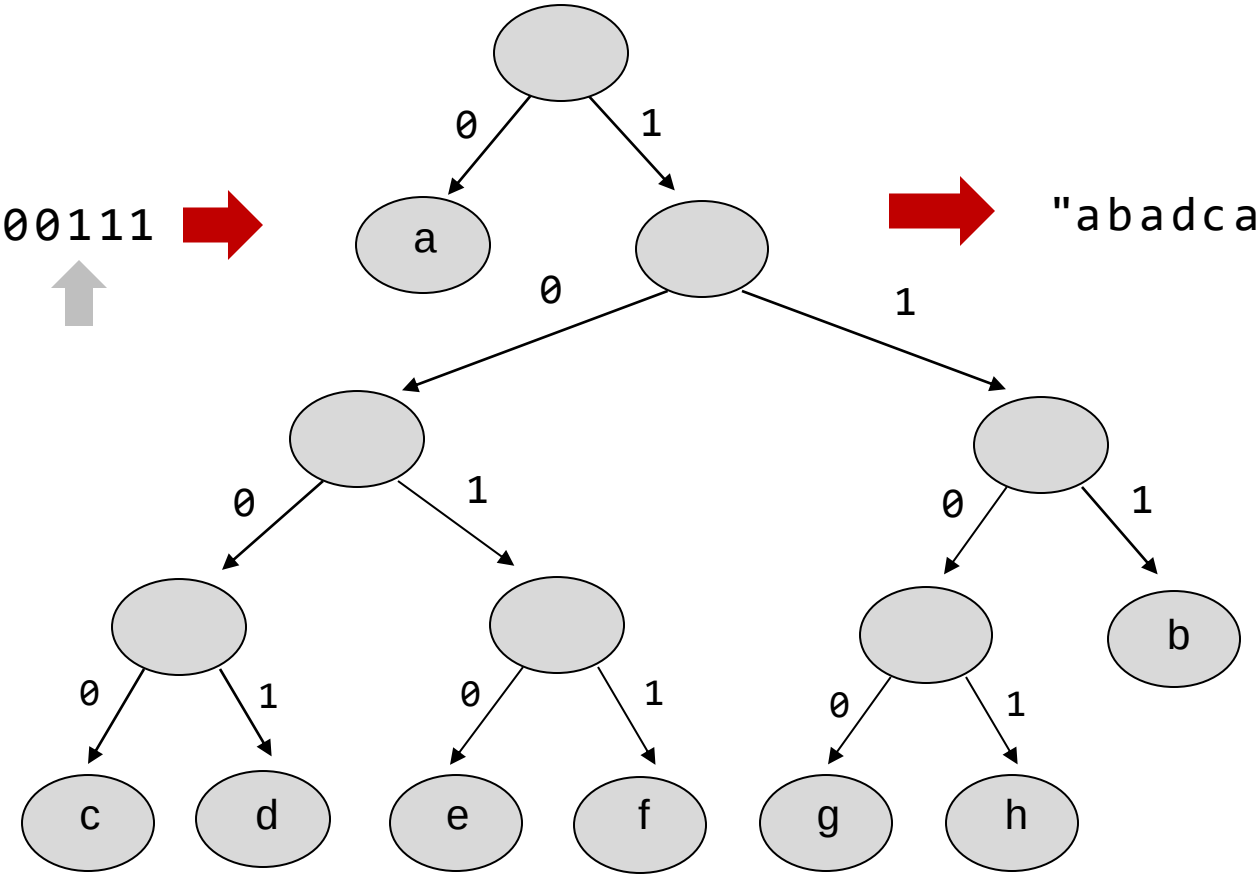


"abadca"

Decompression

Compressed text:

```
01110100110000111
```

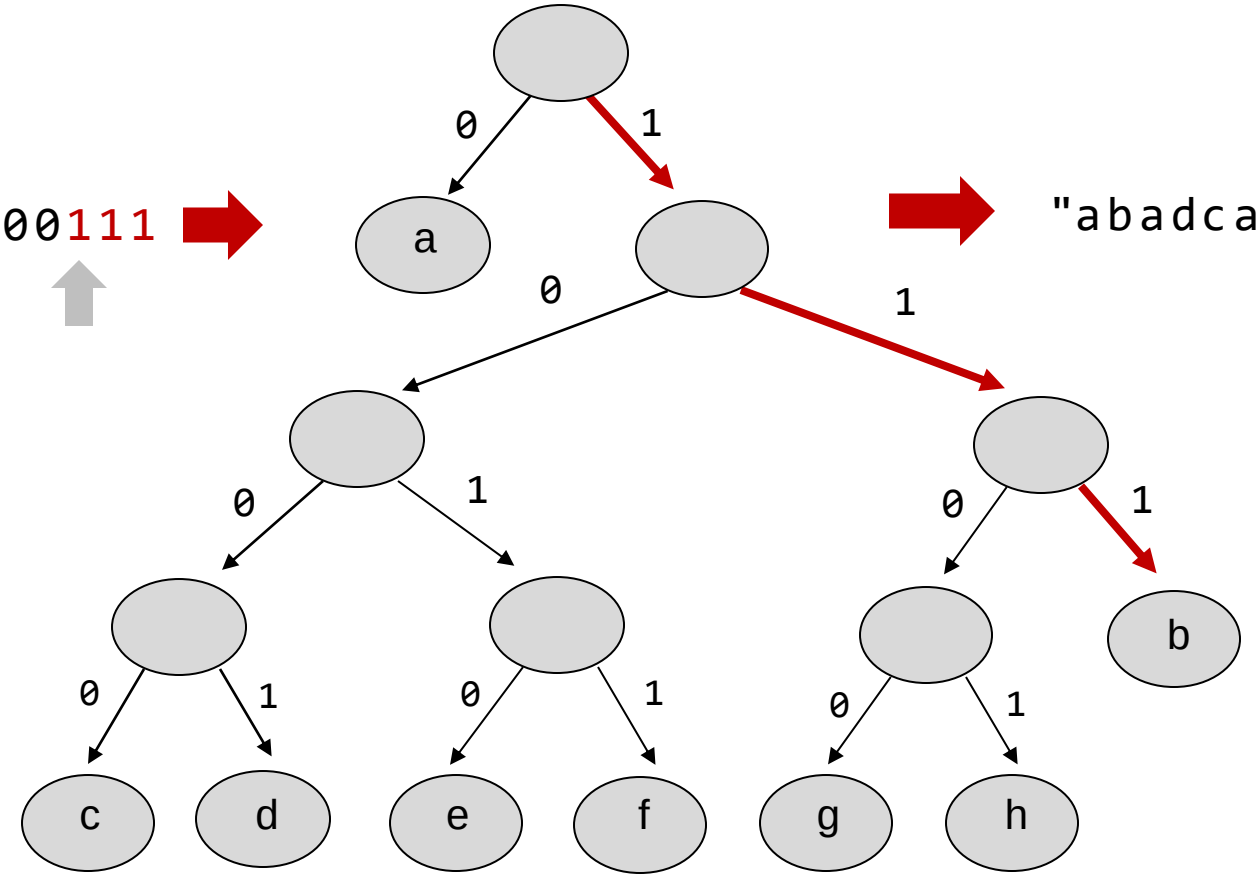


"abadca"

Decompression

Compressed text:

`01110100110000111`

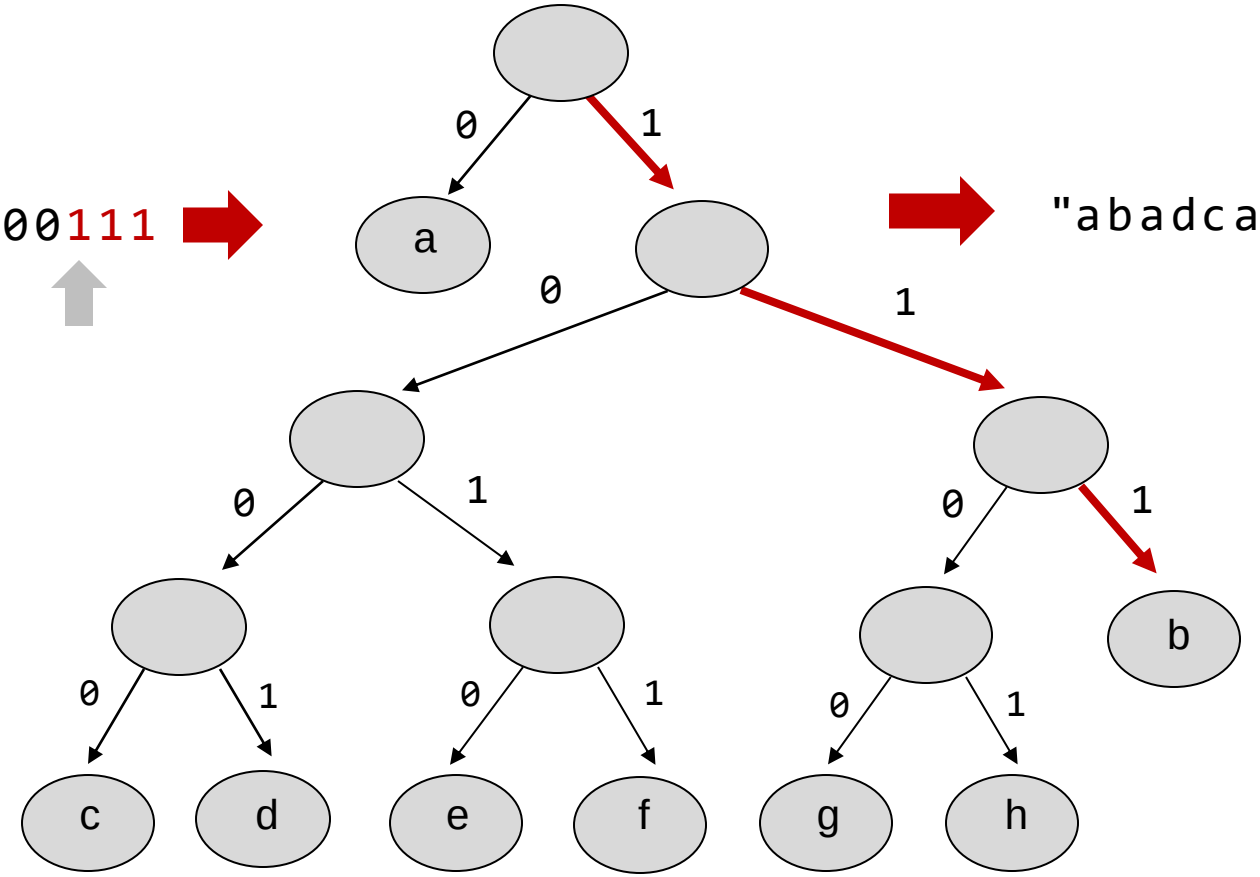


"abadca"

Decompression

Compressed text:

`01110100110000111`

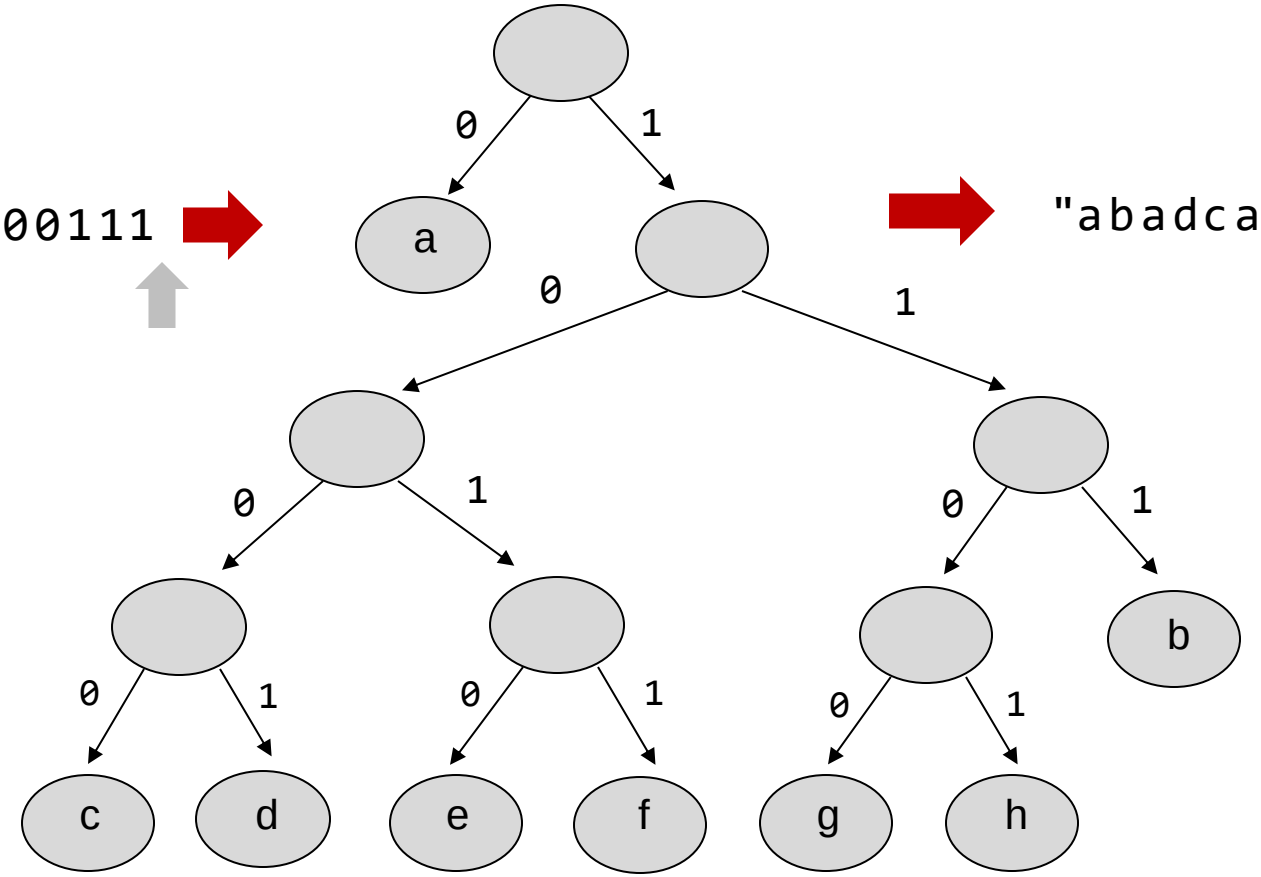


"abadcab"

Decompression

Compressed text:

`01110100110000111`

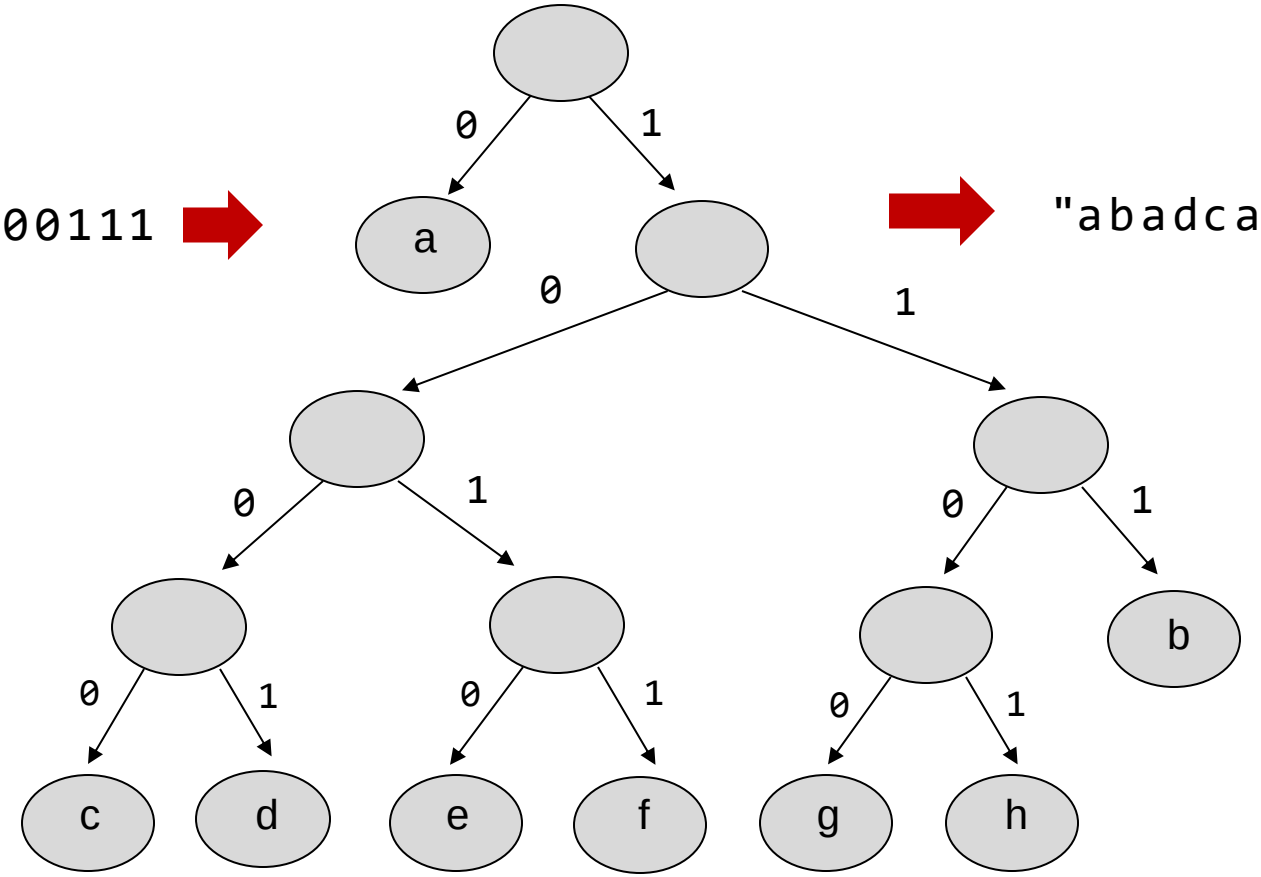


"abadcab"

Decompression

Compressed text:

`0111010011000111`



`"abadcab"`

Observations

- **Prefix free:** No need to separate codes with markers
- **Lossless**: `decompress(compress("abadcab")) = "abadcab"`

Lecture plan

- Trees
- Compression: Algorithms
- Compression: Implementation
- Compression: Extensions

Using compression

```
/** Provides text compression and decompression methods.
 *  Before using, the init method must be called. */
public class Comp {

    /** Initializes data structures based on the given letter frequencies. */
    public static void init(double[] freq)

    /** Compresses the given text into a string of 0's and 1's. */
   public static String compress(String text)

    /** Decompresses the given coded text into the original text. */
    public static String decompress(String codedText)
}
```

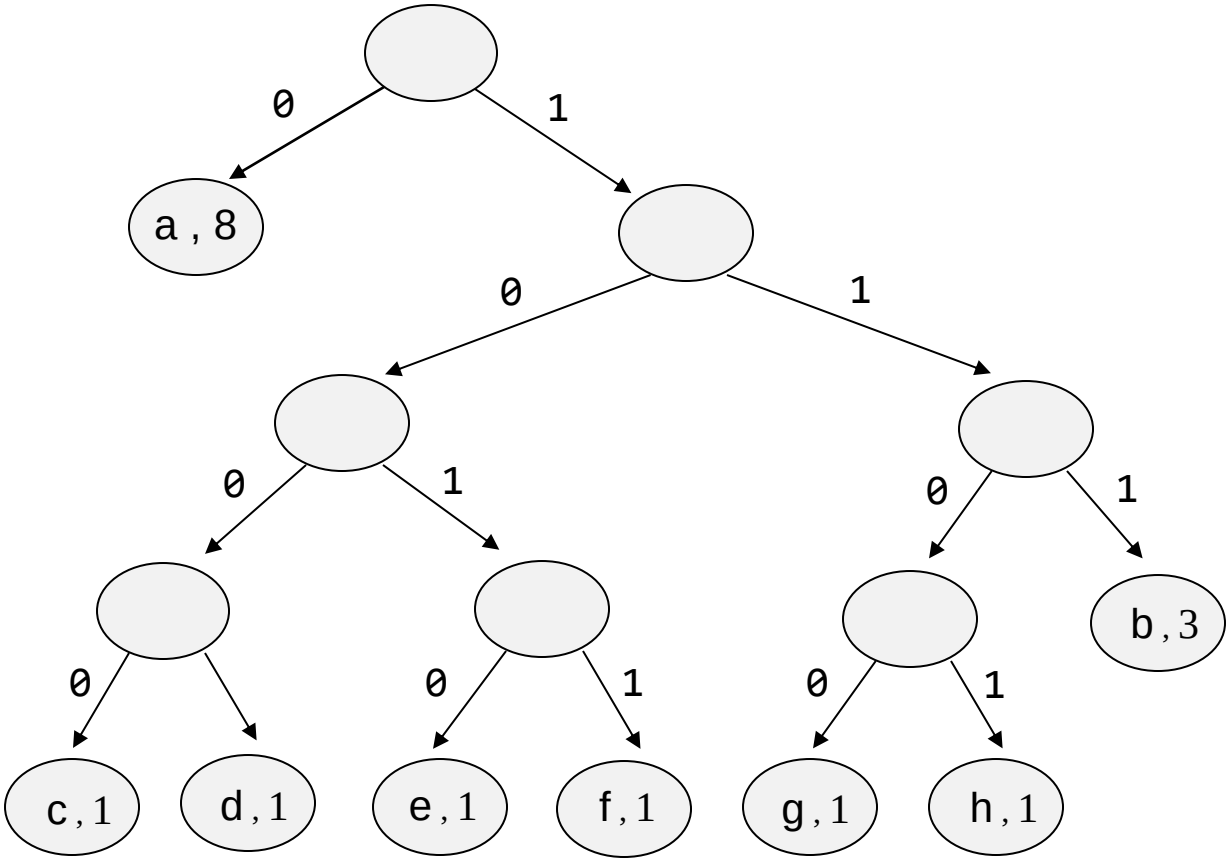
```
// Client code

double[] freq = new double[256];
freq['a'] = 8; freq['b'] = 3; freq['c'] = 1;
freq['d'] = 1; freq['e'] = 1; freq['f'] = 1;
freq['g'] = 1; freq['h'] = 1;

Comp.init(freq);

String text = "abadcab";
System.out.println("Text: " + text);

String compressedText = Comp.compress(text);
System.out.println("Compressed text: " +
                   compressedText);

System.out.println("Decompressed text: " +
                   Comp.decompress(compressedText));
```

output

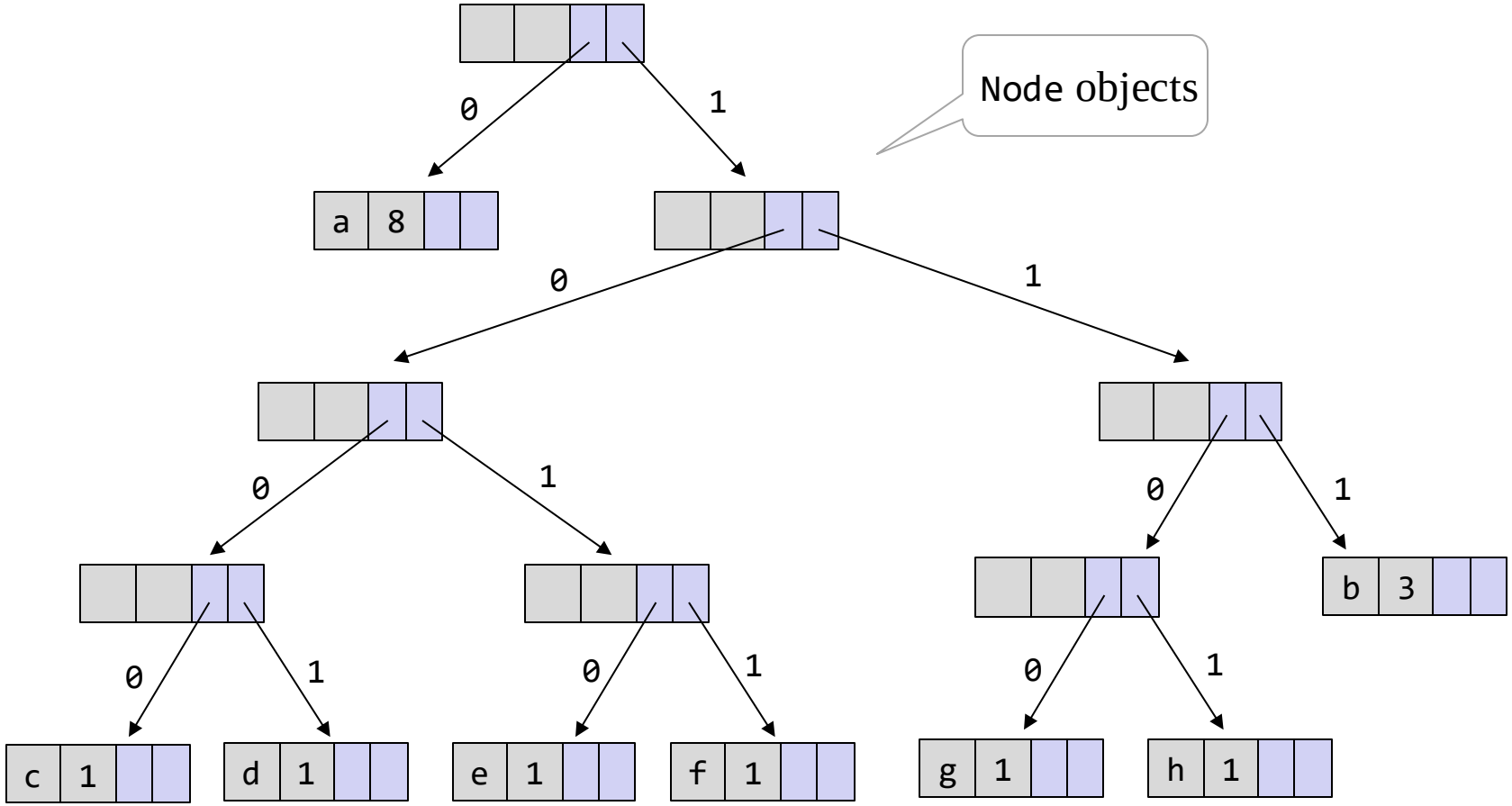
```
Text: abadcab
Compressed text: 01110100110000111
Decompressed text: abadcab
```

lossless

Behing the scene”: Huffman Tree

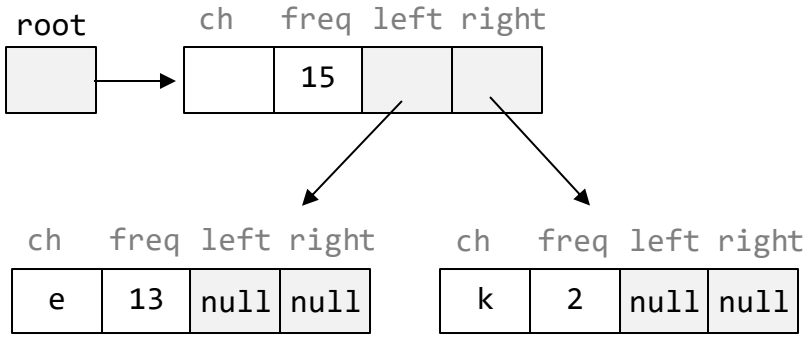


Tree implementation = connected Node objects



Node class

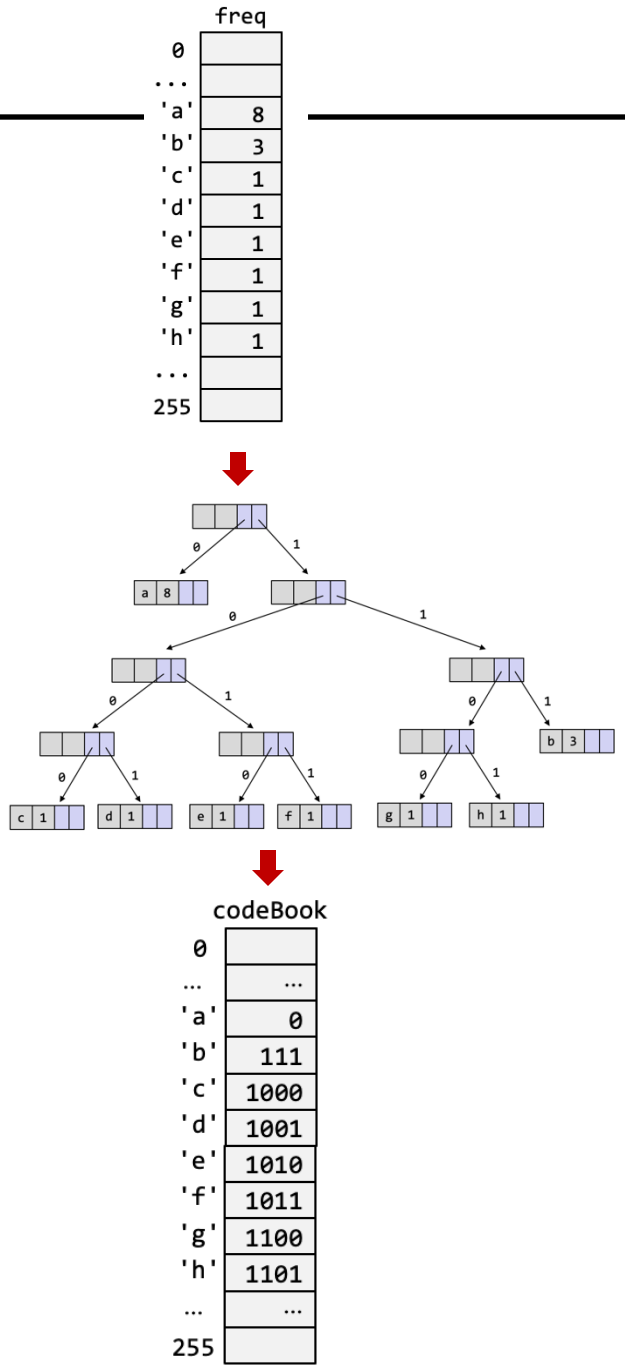
```
/** Represents a node in a Huffman tree. */
public class Node {
    char ch;            // character
    double freq;        // character's frequency
    Node left, right;   // pointers to child nodes

    /** Constructs a branch (interim) node */
    Node(char ch, double freq, Node left, Node right) {
        this.ch = ch;      this.freq = freq;
        this.left = left;  this.right = right;
    }

    /** Constructs a leaf (terminal) node */
    Node(char ch, double freq) {
        this.ch = ch;      this.freq = freq;
        this.left = null;  this.right = null;
    }

    /** Checks if this node is a leaf node */
    boolean isLeaf() {
        return ((left == null) && (right == null));
    }

    /** Prints the leaves below this node, L to R (for testing) */
    // Algorithm: Depth First Search (DFS) tree traversal
    public void print() {
        if (isLeaf()) {
            System.out.println(ch + ": " + freq);
        } else {
            left.print();
            right.print();
        }
    }
}
```



Tree implementation:
connected Node objects

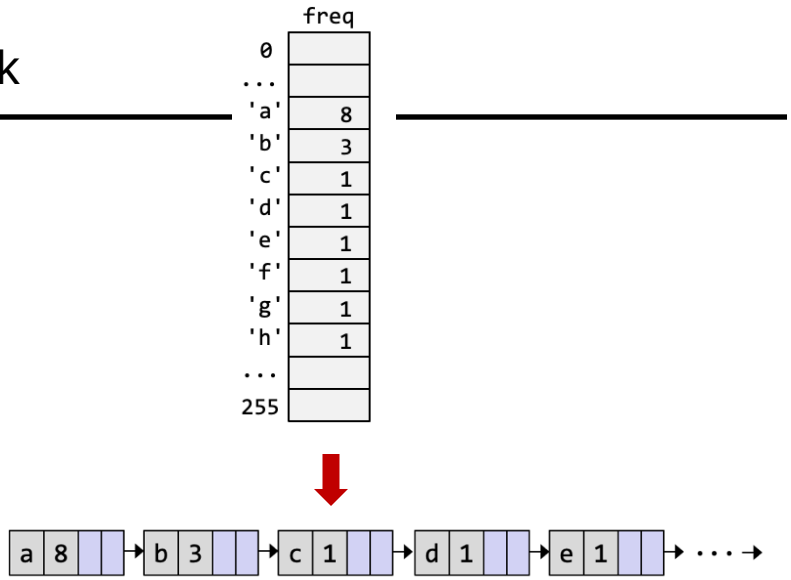
Compression class: Building the code book

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    ...
```



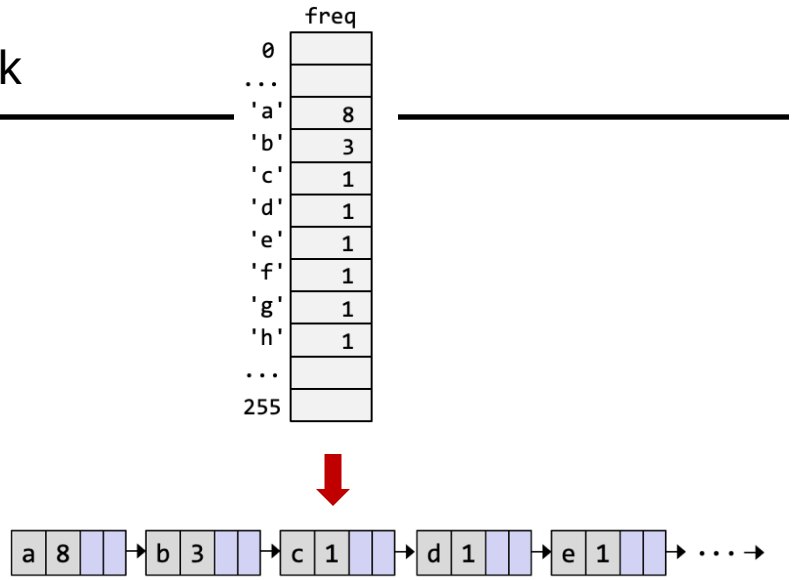
Compression class: Building the code book

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    // Returns a Huffman tree, built from the given letter frequencies. */
    private static Node buildTree(double[] freq) {
        // Builds the initial, flat tree (list of Node elements)
        LinkedList<Node> tree = new LinkedList<>();
        // If a letter frequency is greater than 0, adds a new node
        for (char ch = 0; ch < freq.length; ch++) {
            if (freq[ch] > 0) {
                tree.add(new Node(ch, freq[ch]));
            }
        }

        //// Continues on next slide
```



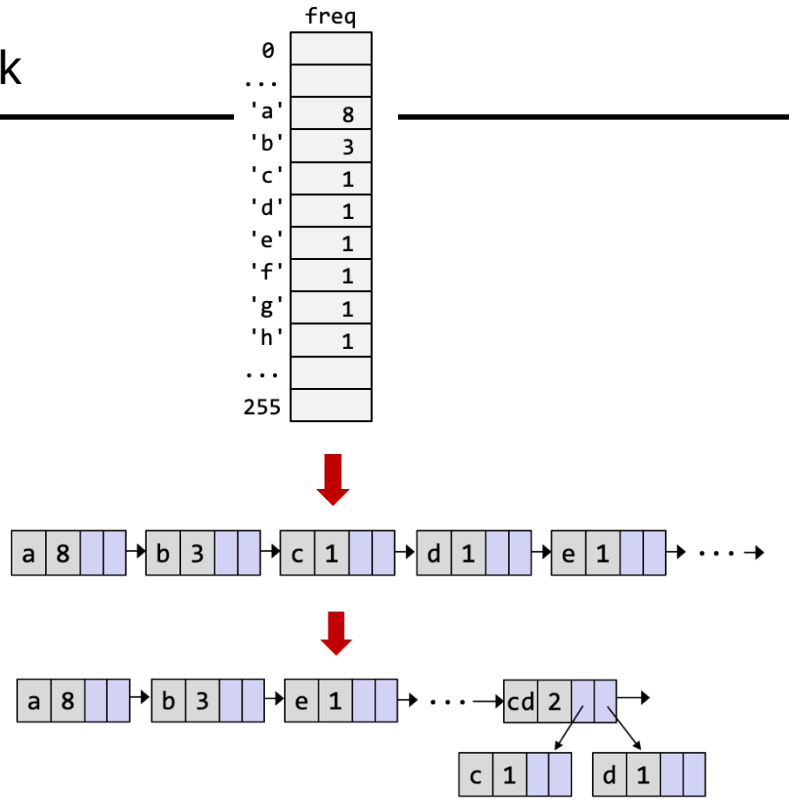
Compression class: Building the code book

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    // Returns a Huffman tree, built from the given letter frequencies. */
    private static Node buildTree(double[] freq) {
        // Builds the initial, flat tree (code omitted)
        ...
```



Compression class: Building the code book

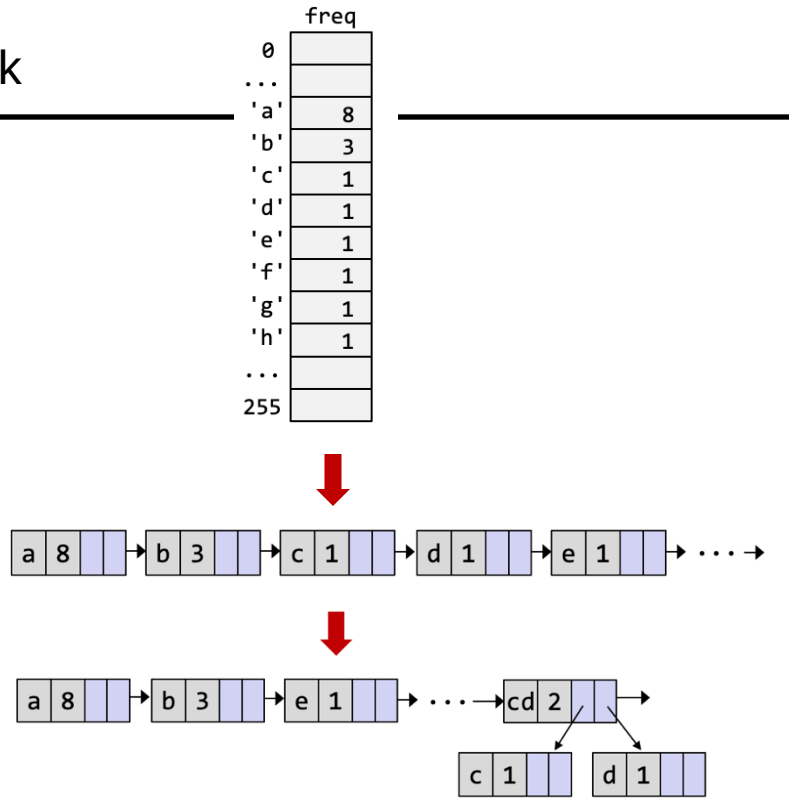
```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    // Returns a Huffman tree, built from the given letter frequencies. */
    private static Node buildTree(double[] freq) {
        // Builds the initial, flat tree (code omitted)
        ...
        // Main loop: Removes the two nodes with the minimal frequencies,
        // merges them into a new node, and adds the new node to the tree.
        // Keeps doing this until the tree has only one node.
        while (tree.size() > 1) {
            Node min1 = removeMin(tree);
            Node min2 = removeMin(tree);
            tree.add(new Node(' ', min1.freq + min2.freq,
                              min1, min2));
        }
        return tree.getFirst(); // Returns the root node
    }
```



Compression class: Building the code book

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    // Returns a Huffman tree, built from the given letter frequencies. */
    private static Node buildTree(double[] freq) {
        // Builds the initial, flat tree (code omitted)
        ...
        // Main loop: Removes the two nodes with the minimal frequencies,
        // merges them into a new node, and adds the new node to the tree.
        // Keeps doing this until the tree has only one node.
        while (tree.size() > 1) {
            Node min1 = removeMin(tree);
            Node min2 = removeMin(tree);
            tree.add(new Node(' ', min1.freq + min2.freq,
                              min1, min2));
        }
        return tree.getFirst
    }
```

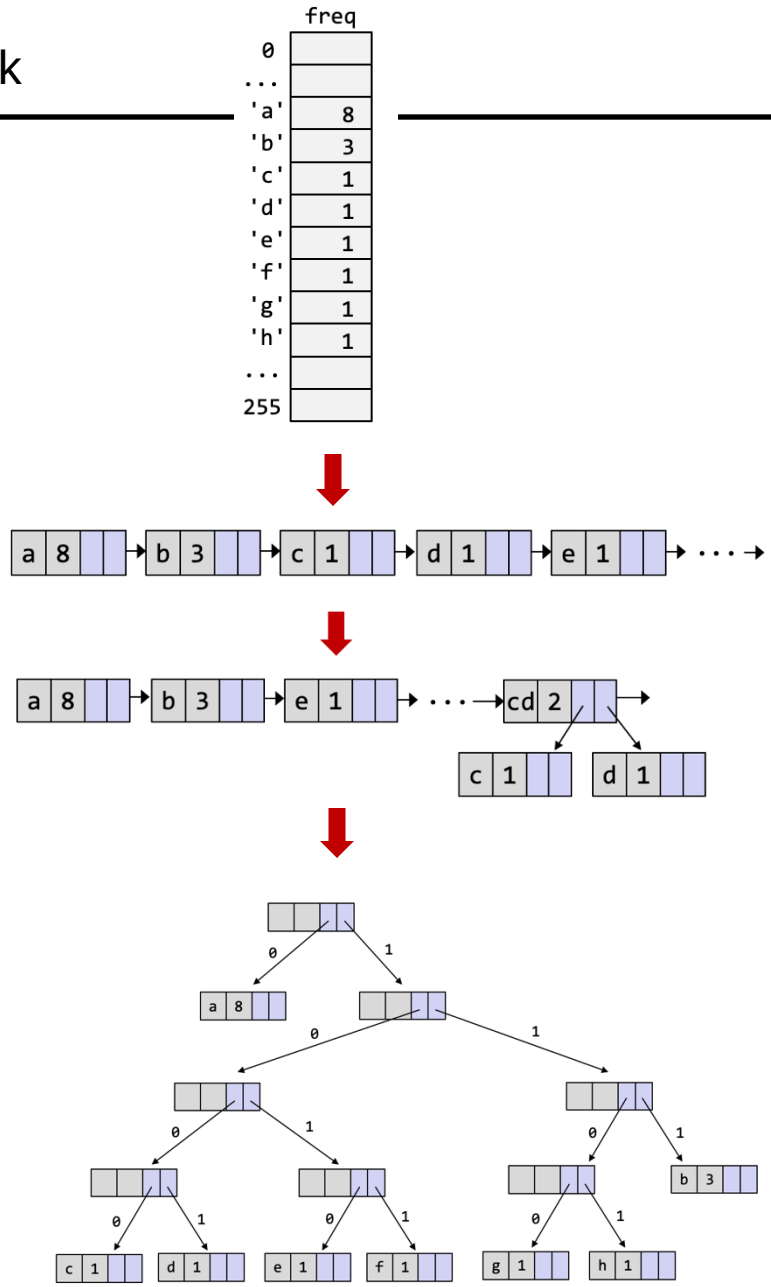
(we can use `for-each` loop since `LinkedList` implements the `Iterable` interface)




```
// Finds the node with the minimal frequency in the list,
// removes this node from the list, and returns it.
private static Node removeMin(LinkedList<Node> list) {
   Node minNode = list.getFirst();
   for (Node node : list)
      if (node.freq < minNode.freq)
         minNode = node;
   list.remove(minNode);
   return minNode;
}
```

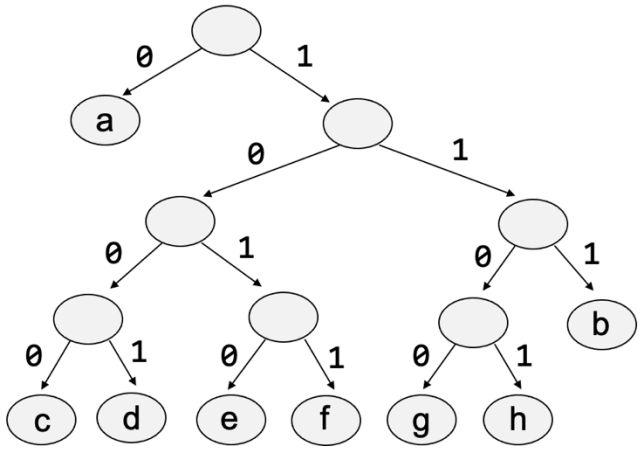
Compression class: Building the code book

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    // Returns a Huffman tree, built from the given letter frequencies. */
    private static Node buildTree(double[] freq) {
        // Builds the initial, flat tree (code omitted)
        ...
        // Main loop: Removes the two nodes with the minimal frequencies,
        // merges them into a new node, and adds the new node to the tree.
        // Keeps doing this until the tree has only one node.
        while (tree.size() > 1) {
            Node min1 = removeMin(tree);
            Node min2 = removeMin(tree);
            tree.add(new Node(' ', min1.freq + min2.freq,
                              min1, min2));
        }
        return tree.getFirst(); // Returns the root node
    }
```



Compression class: Building the code book

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }
    ...
    // Populates the codeBook by assigning binary codes to each character.
    // Side effect: Modifies the codeBook static variable. */
    private static void populateCodeBook(Node node,
                                         String code,
                                         String[] codeBook) {
        // Assigns the codes, using DFS
        if (node.isLeaf()) {
            // When teaching a leaf, the code is complete;
            // Assigns the code to the character, and exits
            codeBook[node.ch] = code;
            return;
        }
        // Builds the code recursively, by going left or right
        populateCodeBook(node.left, code + '0', codeBook);
        populateCodeBook(node.right, code + '1', codeBook);
    }
    ...
```

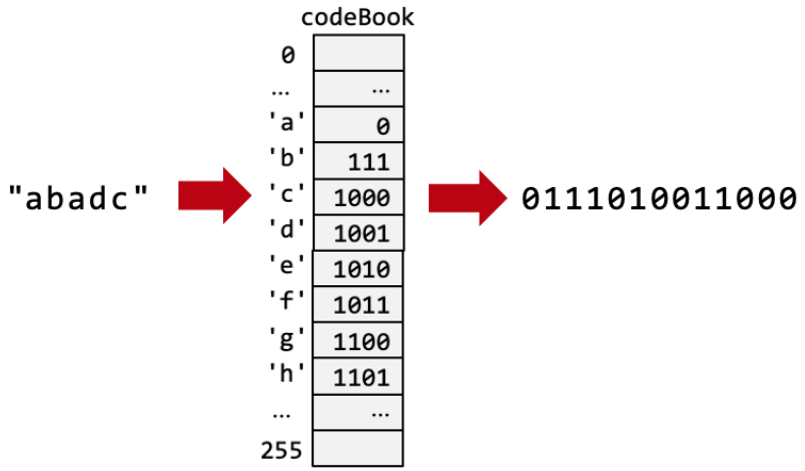


codeBook

0	
...	...
'a'	0
'b'	111
'c'	1000
'd'	1001
'e'	1010
'f'	1011
'g'	1100
'h'	1101
...	...
255	

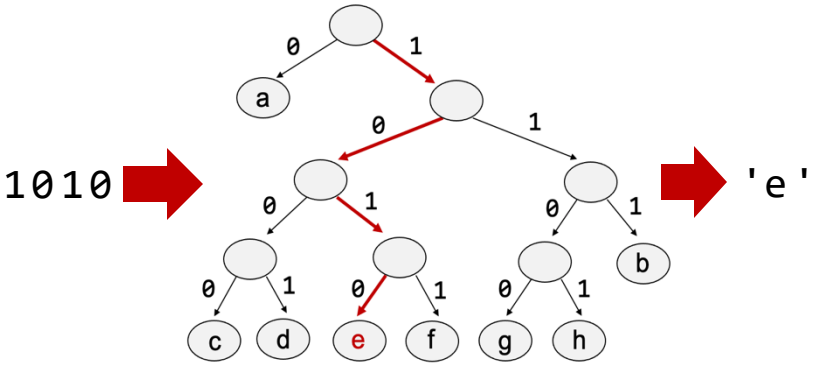
Compression class: Compressing

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    /** Initializes data structures, based on the given letter frequencies. */
    public static void init(double[] freq) {
        // Sets root to the Huffman tree
        root = buildTree(freq);
        // Builds and populates the code book
        codeBook = new String[freq.length];
        populateCodeBook(root, "", codeBook);
    }

    ...

    /** Compresses the given text into a string of 0's and 1's. */
    public static String compress(String text) {
        StringBuilder codedText = new StringBuilder();
        // for each character in the given text, appends its code
        for (int i = 0; i < text.length(); i++) {
            codedText.append(codeBook[text.charAt(i)]);
        }
        return codedText.toString();
    }

    ...
```



Compression class: Decompressing

```
import java.util.LinkedList;

public class Comp {
    static Node root;           // top node
    static String[] codeBook;   // array of character codes

    ...

    /** Decompresses the given coded text into the original text. */
    public static String decompress(String codedText) {
        StringBuilder text = new StringBuilder();
        Node node = root;
        for (int i = 0; i < codedText.length(); i++) {
            char ch = codedText.charAt(i);
            // if 0, go left; if 1, go right
            if (ch == '0') {
                node = node.left;
            } else {
                node = node.right;
            }
            // if leaf node, appends the character
            if (node.isLeaf()) {
                text.append(node.ch);
                // Resets to the root, for the next sequence of 0's and 1's
                node = root;
            }
        }
        // Returns the decompressed text
        return text.toString();
    }
    ...
```



Lecture plan

- Trees
- Compression: Algorithms
- Compression: Implementation
- Compression: Extensions

Extensions

- Instead of encoding single letters (1-grams), we can encode combinations of 2 or more letters (n-grams), or complete words;
- In English, the 100 most commonly used words make up 50% of the text;
- The resulting tree will be much deeper, but more efficient in terms of *average* compression time.

Code optimality

Data: $\{c_1, c_2, \ldots, c_n\}$ characters with frequencies $\{q_1, q_2, \ldots, q_n\}$

A code E encodes each character with a sequence of bits $E(c_i)$

The *weighted length* of E is defined as $L(E) = \sum_{i=1}^{n} q_i \cdot E(c_i)$

$\tilde{E}$ is an *optimal code* if for any code E we have $L(\tilde{E}) \leq L(E)$

Huffman code is optimal

(proof not given here)

Information theory

The study of quantifying, encoding, and transmitting information *efficiently*, focusing on concepts like entropy, data compression, and error correction.

Applications

Huffman coding is widely used in practice

- File compression: zip / gzip (uses Huffman + LZ algorithm)
- Multimedia: JPEG (0.2 compression ratio), MP3 (0.1), MPEG (0.01)
- Communications systems
- Sensors with limited memory
- DNA / protein coding
- Decision trees
- . . .

Example: Decision trees

Used in many AI systems

Example: Medical diagnosis

Some diagnoses are common, other are rare

Huffman coding is used to assign shorter codes (tree paths) to common diagnoses

Result: Shorter tree lookup (better decision time).