Huffman Decoding

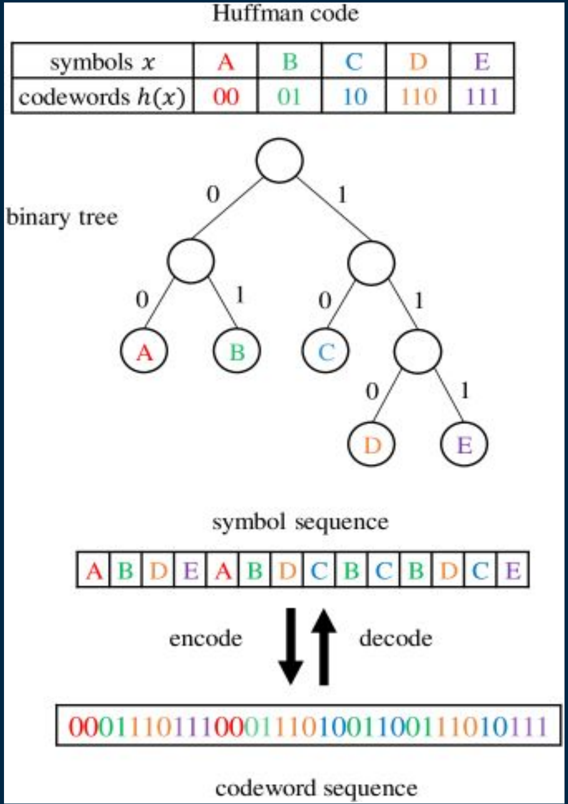
Group 9
Andrew Arteaga
Ana Estrada
Alex Lin

Huffman Coding

Huffman coding is an encoding mechanism by which a variable-length codeword is assigned to each fixed-length input character that is purely based on their frequency of occurrence of the character in the text to be encoded.

Huffman coding has both encoding and decoding process.

GOAL: To try to reduce the total number of bits used without losing any information



Design Techniques

Huffman coding is a data compression algorithm.

- Lossless Compression is a technique that does not lose any data in the compression technique.
- Variable length encoding is where codes are assigned to all characters depending on how frequently they occur in a text.
- Prefix code requires that there is no whole code word in the system that is a prefix of any other code word in the system.
- Priority Queue is a data structure where every item has a priority, An element with high priority is queued before an element with low priority.

One of the main rules of Huffman Code is that no code word is a prefix of another codeword.

Example: [1, 2, 3, 4, 5] is a prefix code
[1, 2, 3, 14, 5] is not a prefix code because 14 has the same start as the value 1.

Huffman Encoding

What is Huffman Encoding?

- It is a greedy approach algorithm that encodes a message into binary form efficiently in terms of space. We use lossless compression in order to avoid losing data during the compression process.
- During this process we assign codewords to each character in the text based on their frequency.

Example: AVADA KEDAVRA = 12 characters

Frequencies: A=5 V=2 D=2 K=1 E=1 R=1

Huffman Encoding

Step 1

List all characters in order from most to least used

Step 2

Place the frequencies of appearance next to each other

Step 3

Combine the two least frequent values together and continue to do so until we’ve created a binary tree

Step 4

We have to go down the tree, 0’s on the left, 1s on the right

Step 5

The codewords will be the concatenation of each value in the tree

Approaches

Divide and Conquer:

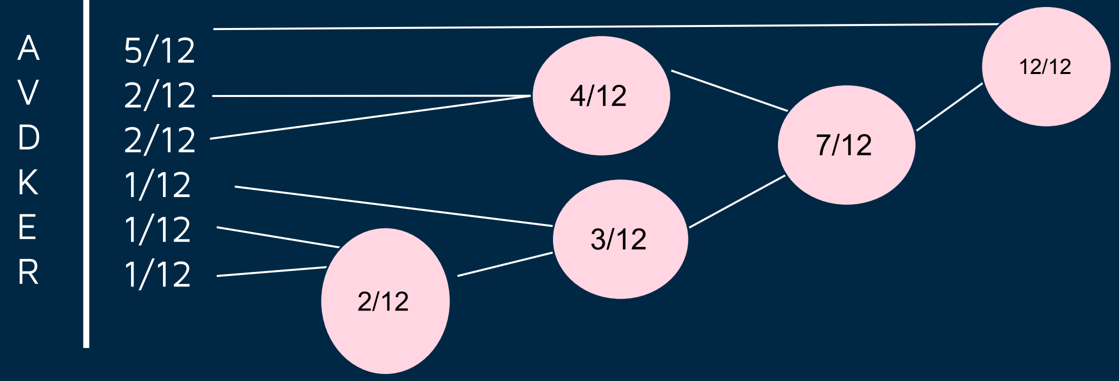
- This approach might have us asking which characters should appear in the left and right subtrees and trying to build the tree from the top down.
- This can lead to an exponential time algorithm

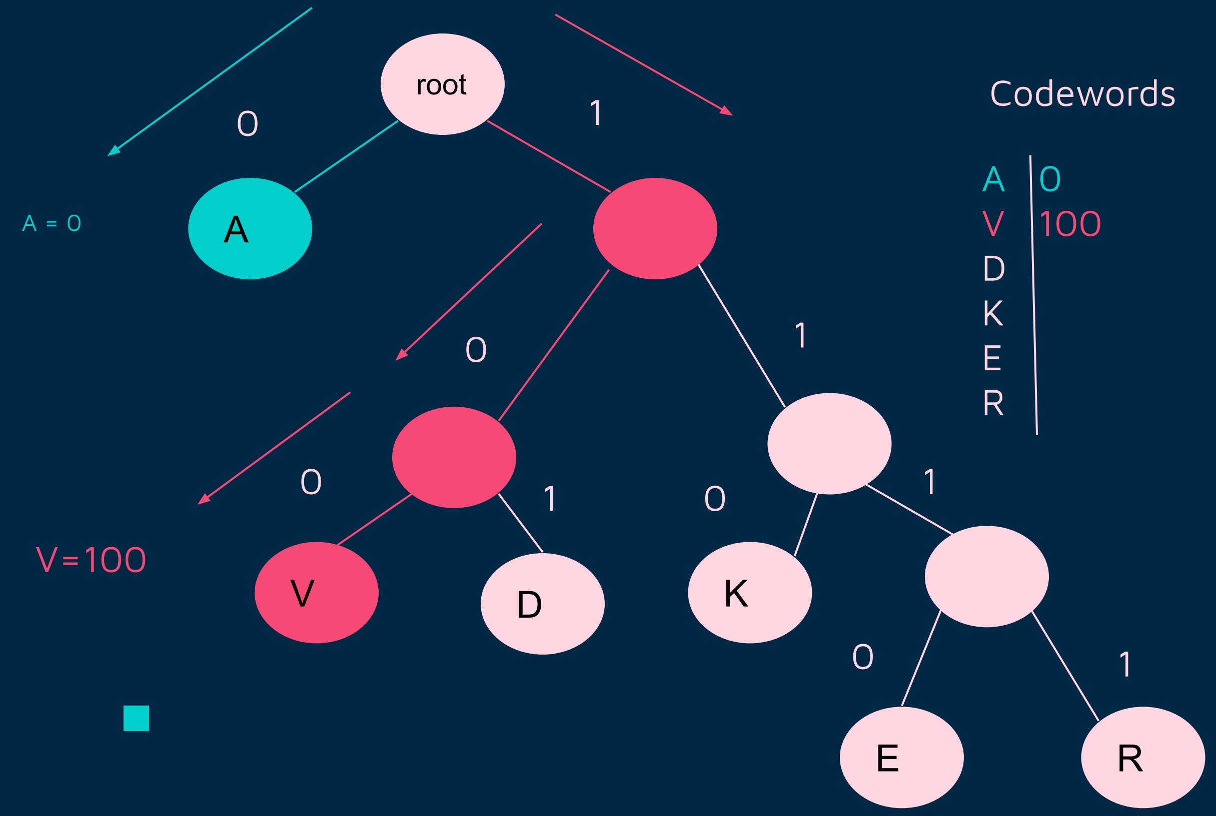
Greedy Approach:

- This is the optimal approach
- It starts with combining the two least weighted nodes into a tree which is assigned the sum of the two leaf nodes.

Example of Huffman Encoding

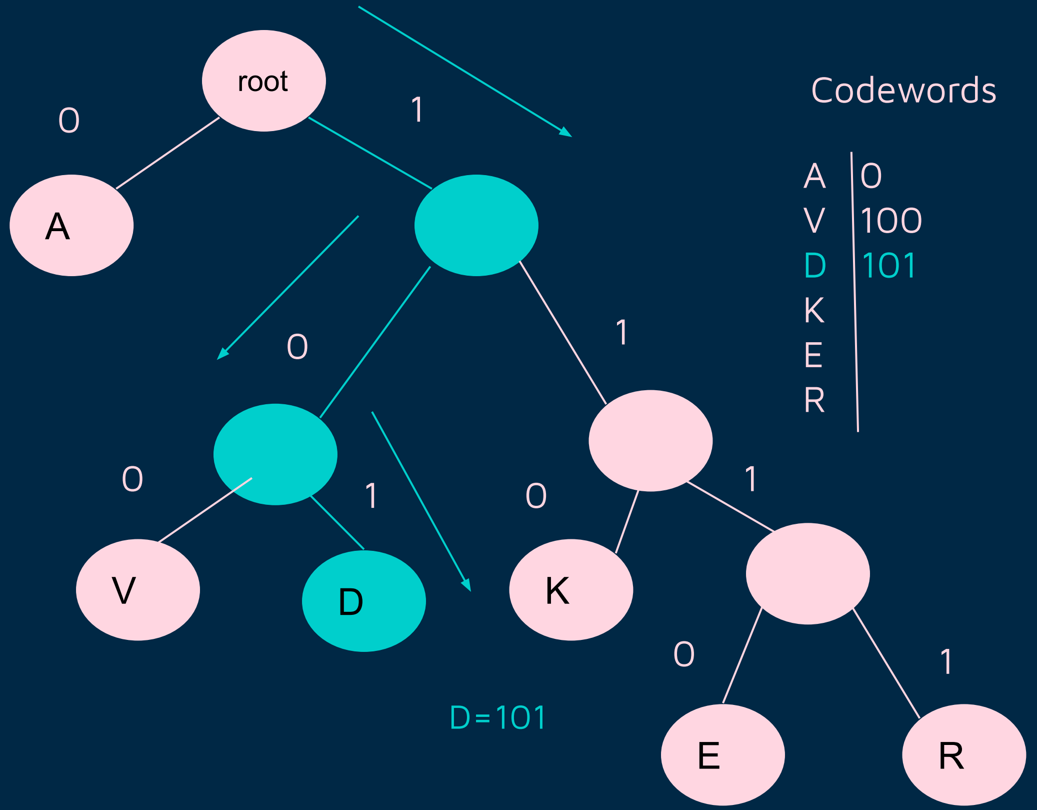
AVADA KEDAVRA = 12 letters





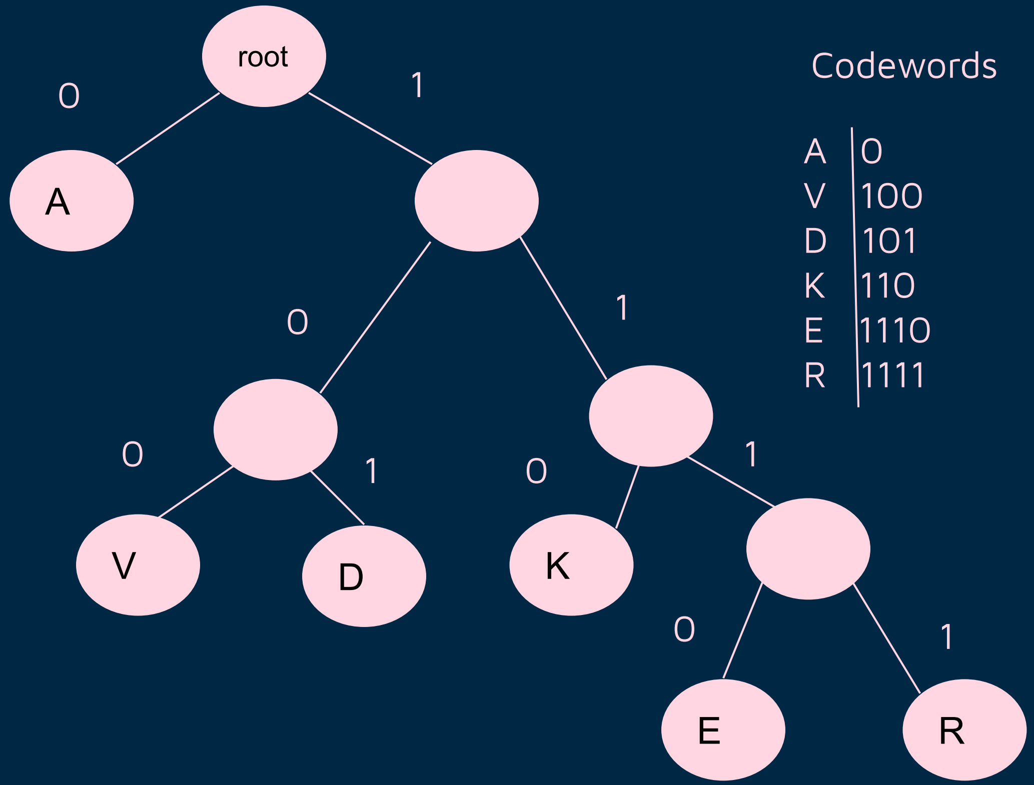
Codewords

A	0
V	100
D	
K	
E	
R	



Codewords

A	0
V	100
D	101
K	
E	
R	



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

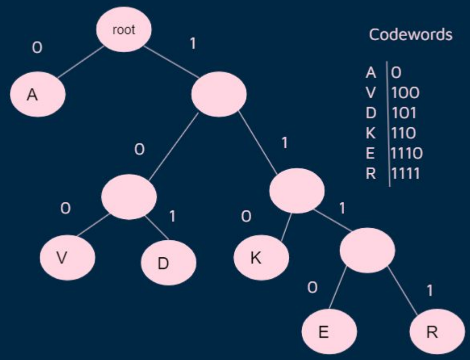
A	V	A	D	A	K	E	D	A	V	R	A	:	Text
0	100	0	101	0	110	1110	101	0	100	1111	0	:	binary string

Huffman Decoding

What is Huffman Decoding?

- It is a greedy algorithm that converts an encoded string to the original string.
- The goal of the Huffman Decoding algorithm is to decode a given codeword to find the corresponding encoded characters against the given Huffman Tree that we obtained during the Huffman Encoding process.

Example: Lets decode this ‘01000101011011101010011110’



Huffman Decoding

Step 1

Start at the root node on our tree.

Step 2

If the current bit in the given data is 0, then move to the left node of the tree

Step 3

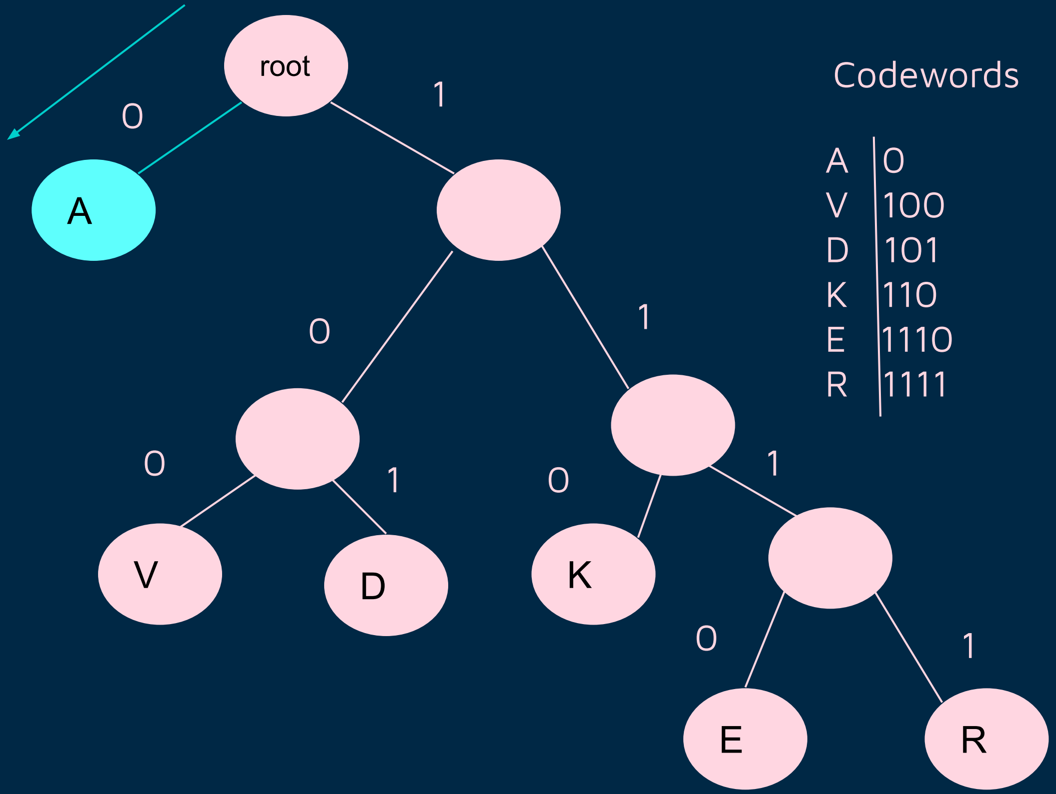
If the current bit in the given data is 1, then move to the right node of the tree

Step 4

During the traversal, if the node is encountered, then print the character of the leaf node

Step 5

Again continue iteration of the encoded data starting at step 1

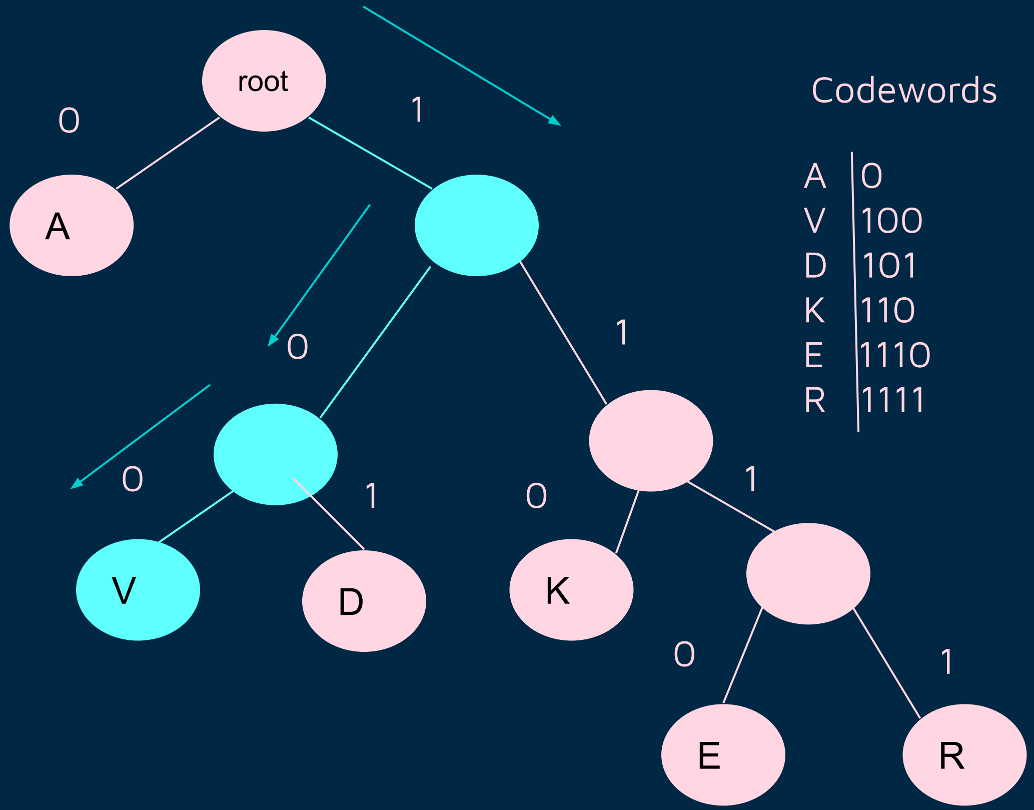


Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110

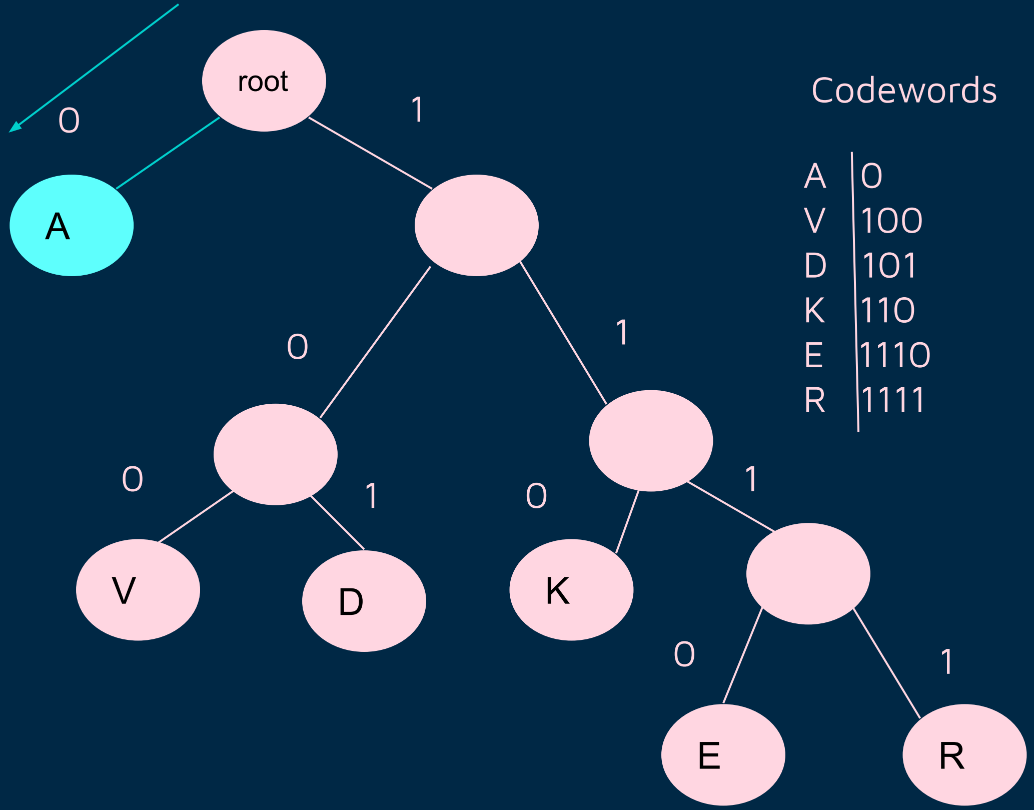
A



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110
AV

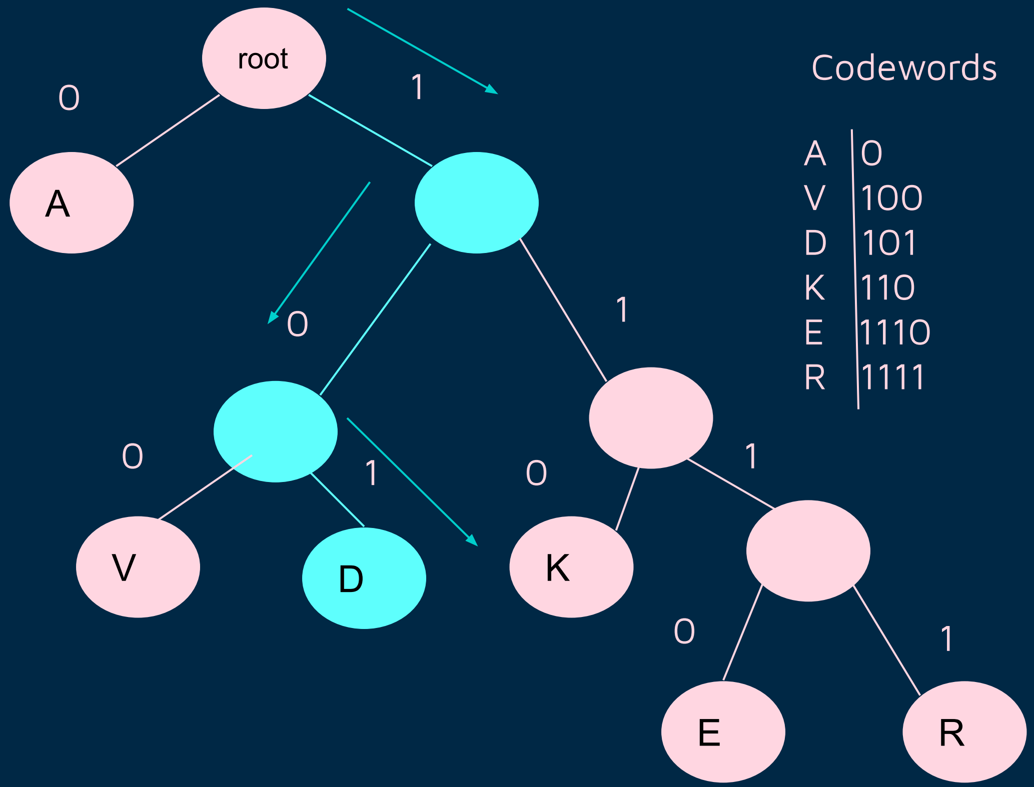


Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110

AVA

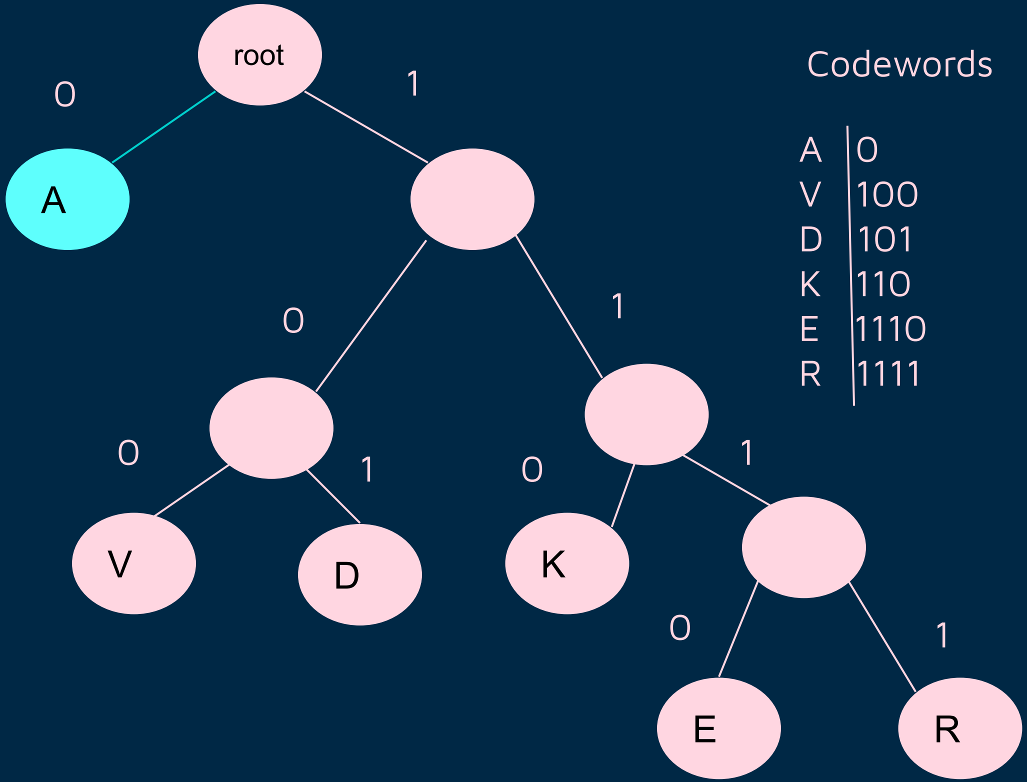


Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

01000101011011101010100111110

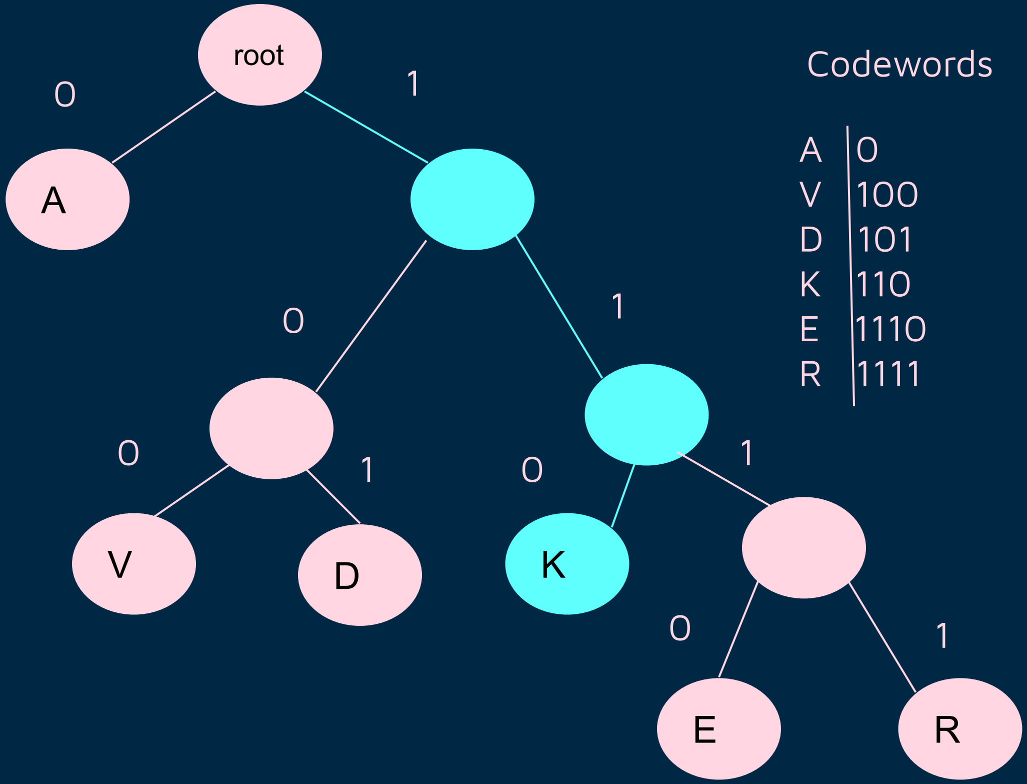
AVAD



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110
AVADA

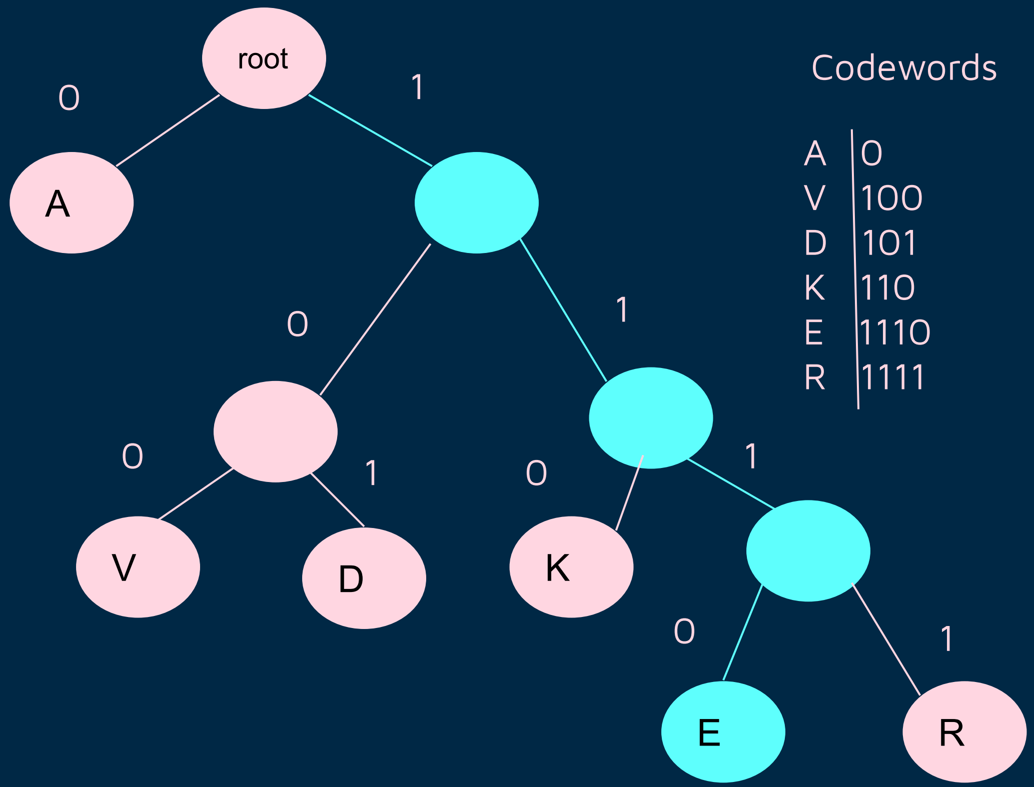


Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

010001010110111010101001111O

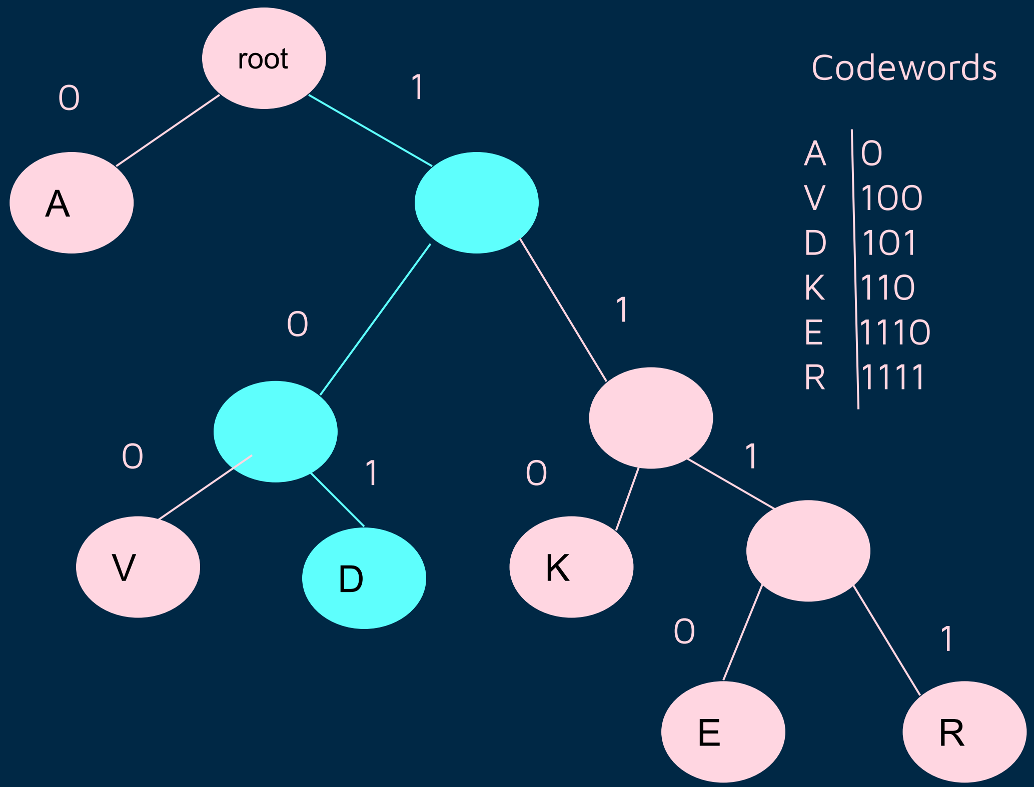
AVADA K



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

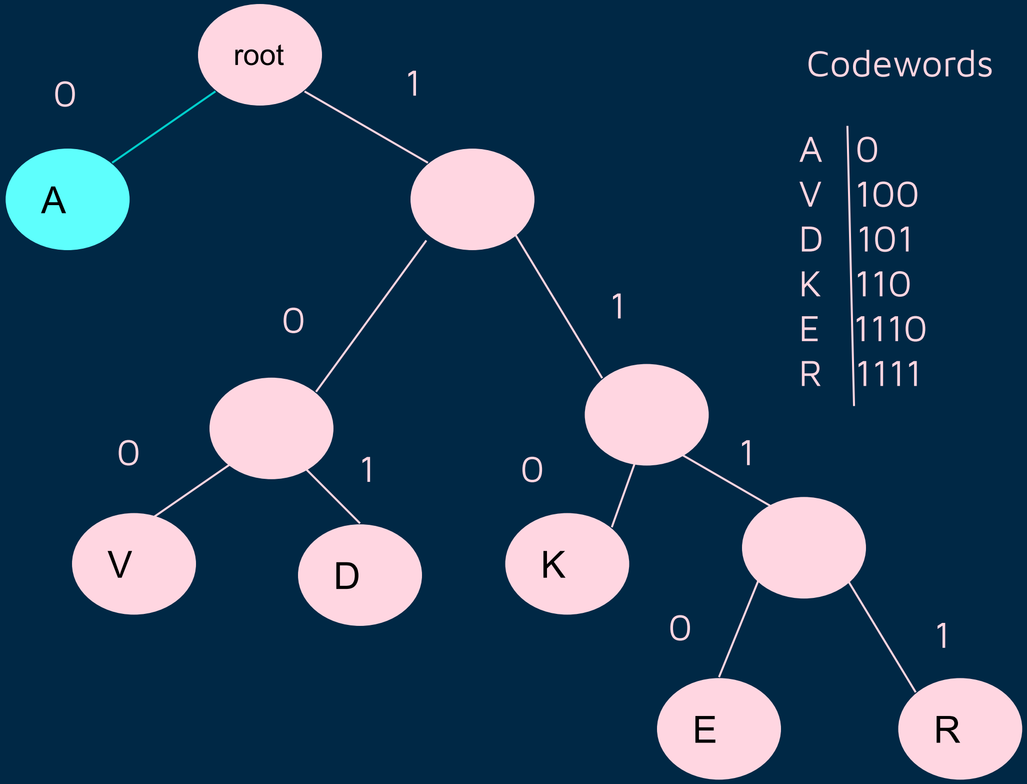
0100010101101110101010011110
AVADA KE



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110
AVADA KED

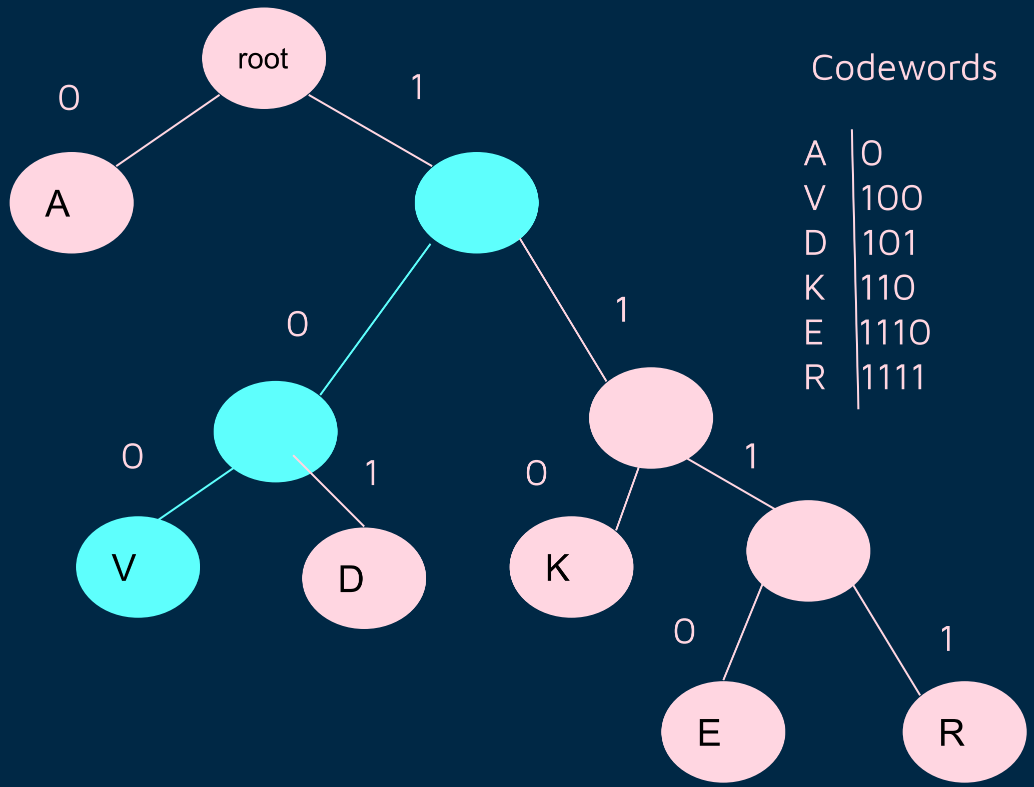


Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

010001010110111010101001110

AVADA KEDA

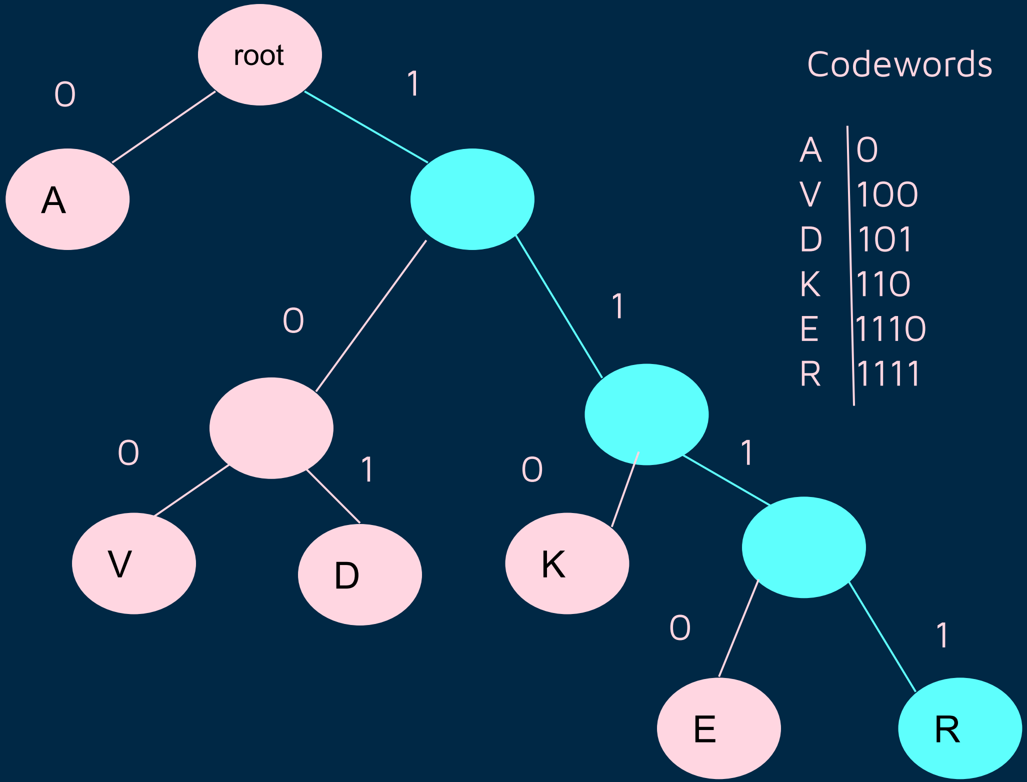


Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110

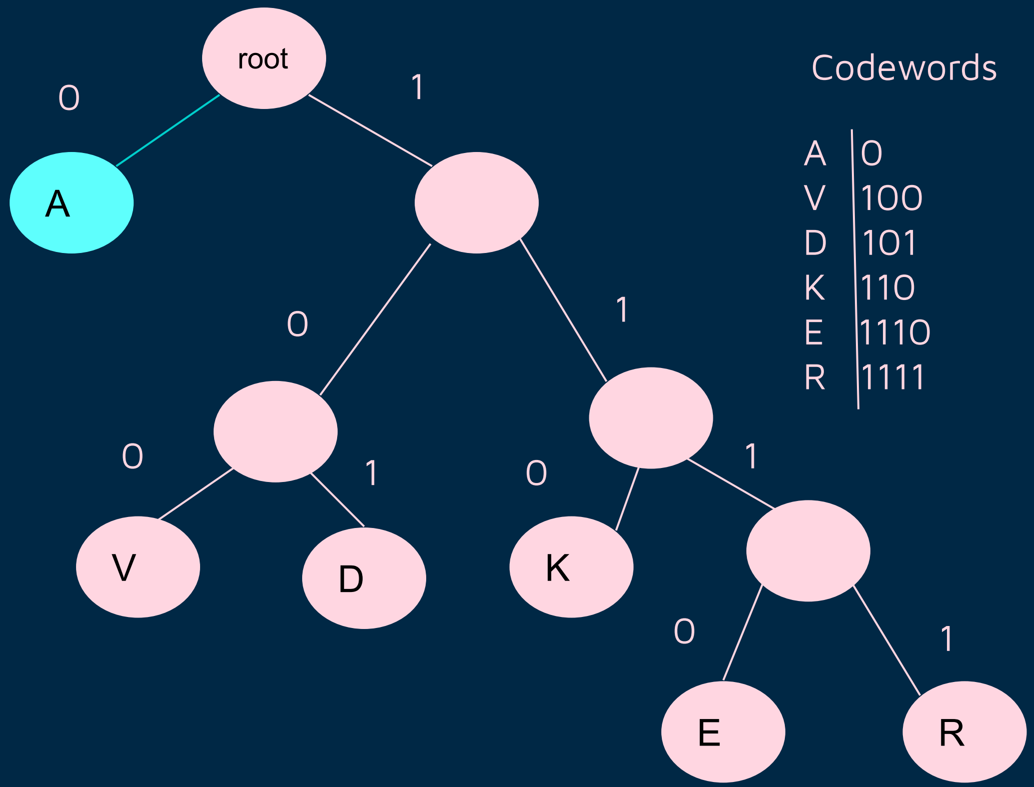
AVADA KEDAV



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110
AVADA KEDAVR



Codewords

A	0
V	100
D	101
K	110
E	1110
R	1111

0100010101101110101010011110
AVADA KEDAVRA

Time Complexity and Running Time

```
Struct nodetype {
      Char symbol; // Value of a character
      Int frequency; // number of times the char is in the file
      nodetype* left;
      nodetype* right;
};
```

Huffman's algorithm

```
For (I = i; i <= n-1; i++) { //solution is obtained when i = n-1
      //Selection procedure
      remove (PQ, p);
      remove (PQ, q);
      r = new nodetype;
      r→left = p;
      r →right = q;
      r → frequency = p → frequency + q → frequency;
      insert (PQ, r);
}
      remove (PQ, r);
      return r;
```

If a priority queue is implemented as hash it can be initialized in O(n) time.

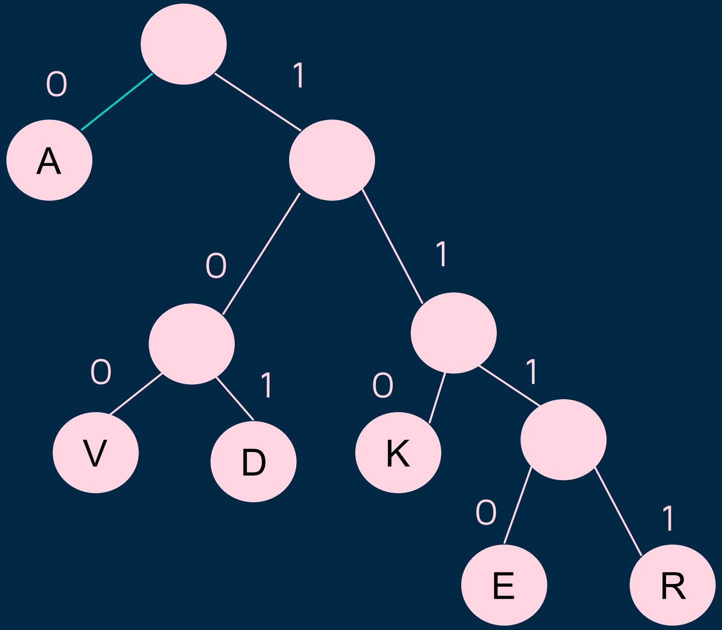
Each operation requires O(logn) time

There are n-1 passes through for-i loop

Total running time is $O((n-1) \log n) = O(n \log n)$

So the algorithm runs in O(nlogn) time.

Time Complexity and Running Time



A Huffman code is closely related to a binary tree, however unlike a binary tree a Huffman code only stores its information in the leaf nodes.

The Huffman tree is treated as the binary tree associated with minimum external path weight that means, the one associated with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to construct a tree with the minimum external path weight.

Time Complexity and Running Time

Worst time complexity for Huffman's algorithm

*"The worst case for Huffman coding can happen when **the probability of the most likely symbol far exceeds 2^{-1} = 0.5**, making the upper limit of inefficiency unbounded. These situations often respond well to a form of blocking called **run-length encoding**."*

This happens when a few letters in the alphabet are much more prevalent than others. (i.e. E **is** the most commonly used **letter** of the **alphabet**, and T **is more common** as the **first letter** of a word).

<u>The reason</u>:

The Huffman algorithm considers the two least frequent elements recursively as the sibling leaves of maximum depth in code tree. The Fibonacci sequence as frequencies list. It is defined to satisfy F(n) + F(n+1) = F(n+2). The resulting tree will be the most unbalanced one, being a full binary tree.

This is why run length coding or other forms of compression are usually applied prior to Huffman coding.
It eliminates many of the worst case scenarios for Huffman coding.
If it turns out the run length encoding is suboptimal it can be skipped

Time Complexity and Running Time

What is run length encoding (RLE)

RLE is a simple form of lossless data compression that runs on sequences with the same value occurring many consecutive times. It encodes the sequence to store only a single value and its count.

`AAABCCCCDD`

With a run−length encoding (RLE) data compression algorithm applied to the above hypothetical scan line, it can be rendered as `3A 1B 4C 2D`.

Demonstration

Huffman Encoding

- Given a string/text
- We iterate through the string, storing each new character encountered into a hashmap or increasing the value of a character in the hashmap that was already encountered
- Then we iterate through the hashmap creating a node that holds a symbol/char and its frequency and add it to a priority queue
- Next we make a binary tree, by iterating through the priority queue and removing the first two values in it, then making them the children of a new node, this new node is then placed back into the priority queue
- We traverse the newly created tree recursively to find the prefix codes of each character, starting the from the root, we recursively call the method to check the left and right side of the node, a 0(left) or 1(right) gets added to a string
- Every time a leaf node is reached, the string value is stored in a hashmap along with the symbol/char of that node
- Lastly, we iterate through the original string and add the prefix code of each character to a new encoded binary string

Huffman Encoding

```
private static Map<Character, Integer> getFrequencies(String test) {
        Map<Character, Integer> hm = new HashMap<>();

        for(int i = 0; i < test.length(); i++) {
            if (!hm.containsKey(test.charAt(i))) {  // check if character is in the hashmap
                hm.put(test.charAt(i), 1);          // store it in hashmap
            }
            else {
                hm.put(test.charAt(i), hm.get(test.charAt(i)) + 1); // increase the value if already in
hashmap
            }
        }

        return hm;
    }
```

```
String test = "Huffman Decoding";
```

Huffman Encoding

```
private static PriorityQueue<Node> makePriorityQueue(Map<Character, Integer> hm) {
        PriorityQueue<Node> pq = new PriorityQueue<>();

        for(char key : hm.keySet()) {
            Node node = new Node();     // create a new node
            node.symbol = key;          // store the key from hashmap in the symbol of node
            node.frequency = hm.get(key);   // store the value of the key in the frequency of the node
            node.left = null;
            node.right = null;
            pq.add(node);       // add the node to the priority queue
        }

        return pq;
    }
```

Huffman Encoding

```
private static Node makeTree(PriorityQueue<Node> pq, int n) {
        Node r = new Node();

        for(int i = 0; i < n-1; i++) {
            Node p = pq.poll();     // get the first node in the queue
            Node q = pq.poll();     // get the second node in the queue
            r = new Node();         // create a new node and store the p as left and q as right
            r.left = p;
            r.right = q;
            r.frequency = p.frequency + q.frequency;    // frequency is the sum of p and q
            pq.add(r);      // add back into the queue
        }

        pq.poll();
        return r;
    }
```

Huffman Encoding

```
private static void getPrefixCodes(Map<Character, String> prefixCodes, Node root, String prefix) {
        if(root != null) {  // check if root is null
            if(root.left == null && root.right == null) {   // check if it node is a leaf
                prefixCodes.put(root.symbol, prefix);       // if it is a leaf add it the prefix to the
hashmap
            }
            else {
                prefix += '0';      // add a 0 to the prefix
                getPrefixCodes(prefixCodes, root.left, prefix);
                prefix = prefix.substring(0, prefix.length()-1);

                prefix += '1';      // add a 1 to the prefix
                getPrefixCodes(prefixCodes, root.right, prefix);
                prefix = prefix.substring(0, prefix.length()-1);
            }
        }
    }
```

Huffman Encoding

```
private static String encode(String test, Map<Character, String> prefixCodes) {
        String encoded = "";

        for(int i = 0; i < test.length(); i++) {    // iterate through original string
            encoded += prefixCodes.get(test.charAt(i)); // add the prefix code of each character to the
encoded string
        }

        return encoded;
    }
```

Huffman Decoding

- Given an encoded binary string
- Given every character and its frequency for the original non-encoded string
- A node is created to store the character and frequency, which is then placed into a priority queue
- We create a binary tree using the same process as in huffman encoding
- Next we need to iterate through the binary string
- We create a temporary node that is a copy of the root node, we then read the first character of the binary string, if it is a 0 the temp node gets the value of its left child, and if it is a 1, we get the value of the right child
- We check if its new children are both null to see if it is a leaf node, if it is the symbol of that node is added to the decoded string and the temp node equals the root node again
- Otherwise, we keep traversing the tree until we hit a leaf node to read or we have read all characters in the binary string

Huffman Decoding

```java
private static PriorityQueue<Node> makePriorityQueue(char[] symbols, int[] frequencies, int n) {
        PriorityQueue<Node> pq = new PriorityQueue<>();

        for(int i = 0; i < n; i++) {
            Node node = new Node();     // create a new node
            node.symbol = symbols[i];   // store the symbol at i in node
            node.frequency = frequencies[i];    // store the frequency at i in node
            node.left = null;
            node.right = null;
            pq.add(node);       // add the node to the priority queue
        }

        return pq;
    }
```

```java
String code = "0101001100000010001101101110011110010011111101001011010100";
        char[] symbols = {' ', 'a', 'c', 'D', 'd', 'e', 'f', 'g', 'H', 'i', 'm', 'n', 'o', 'u'};
        int[] frequencies = {1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1};
```

Huffman Decoding

```
private static Node makeTree(PriorityQueue<Node> pq, int n) {
        Node r = new Node();

        for(int i = 0; i < n-1; i++) {
            Node p = pq.poll();     // get the first node in the queue
            Node q = pq.poll();     // get the second node in the queue
            r = new Node();         // create a new node and store the p as left and q as right
            r.left = p;
            r.right = q;
            r.frequency = p.frequency + q.frequency;    // frequency is the sum of p and q
            pq.add(r);       // add back into the queue
        }

        pq.poll();
        return r;
    }
```

Huffman Decoding

```
private static String decode(String encoded, Node root) {
        String decoded = "";
        Node temp = root;
        for(int i = 0; i < encoded.length(); i++) {
            int binary = Integer.parseInt(String.valueOf(encoded.charAt(i)));
            if(binary == 0)
                temp = temp.left;
                if(temp.left == null && temp.right == null)
                    decoded += temp.symbol;
                    temp = root;
            if(binary == 1)
                temp = temp.right;
                if(temp.left == null && temp.right == null)
                    decoded += temp.symbol;
                    temp = root;
        }
        return decoded;
    }
```

Real World Application

Huffman coding is a commonly used algorithm in the real world. It is still widely used today, although it is mostly used as a basis to other compression methods. The most useful aspect of this algorithm is the prefix codes.

- ZIP, GZIP, PKZIP, BZIP2
- JPEG and PNG
- MP3

Pros and Cons

Pros:

- It results in saving a lot of storage space, since the binary codes generated are variable in length
- The binary codes generated are prefix free

Cons:

- This process is slower
- It is difficult for the decoding software to detect whether the encoded data is corrupt
- Lossless techniques are unsuitable for encoding and decoding digital images

Do you have any questions?

THANKS

RESOURCES

- https://www.hackerrank.com/challenges/tree-huffman-decoding/problem
- https://medium.com/hackerrank-algorithms/tree-huffman-decoding-fd4f973d1f58
- https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art007
- https://en.wikipedia.org/wiki/Huffman_coding
- https://www.youtube.com/watch?v=SIs8zdGU4cQ