CE100 Algorithms and Programming II

Huffman Coding

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-9 (Huffman Coding)

#### Spring Semester, 2021-2022

Download [DOC](ce100-week-9-huffman.tr.md_doc.pdf), [SLIDE](ce100-week-9-huffman.tr.md_slide.pdf), [PPTX](ce100-week-9-huffman.tr.md_slide.pptx)

## Huffman Coding

## Outline

* Heap Data Structure (Review Week-4)
* Heap Sort (Review Week-4)
* Huffman Coding

## **Huffman Codes**

## Huffman Codes for Compression

* Widely used and very effective for data compression
* Savings of 20% - 90% typical
  + (depending on the characteristics of the data)
* **In summary:** Huffman’s greedy algorithm uses a **table of frequencies** of character occurrences to build up an optimal way of **representing each character as a binary string**.

## Binary String Representation - **Example**

* Consider a data file with:
  + 100K characters
  + Each character is one of
* Frequency of each character in the file:
  + **frequency:**
* **Binary character code:** Each character is represented by a unique binary string.
* **Intuition:**
  + Frequent characters shorter codewords
  + Infrequent characters longer codewords

## Binary String Representation - **Example**

* How many total bits needed for **fixed-length** codewords?
* How many total bits needed for **variable-length(1)** codewords?
* How many total bits needed for **variable-length(2)** codewords?

## Prefix Codes

* **Prefix codes:** No codeword is also a prefix of some other codeword
* **Example:**
* It can be shown that:
  + Optimal data compression is achievable with a **prefix code**
* In other words, optimality is not lost due to **prefix-code** restriction.

## Prefix Codes: Encoding

* **Encoding:** Concatenate the codewords representing each character of the file
* **Example:** Encode file “abc” using the codewords above
* **Note:** “.” denotes the concatenation operation. It is just for illustration purposes, and does not exist in the encoded string.

## Prefix Codes: Decoding

* Decoding is quite simple with a prefix code
* The first codeword in an encoded file is unambiguous
  + *because no codeword is a prefix of any other*
* **Decoding algorithm:**
  + Identify the initial codeword
  + Translate it back to the original character
  + Remove it from the encoded file
  + Repeat the decoding process on the remainder of the encoded file.

## Prefix Codes: Decoding - Example

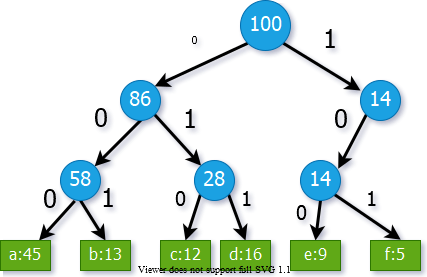
* Example: Decode encoded file

## Prefix Codes

* Convenient representation for the prefix code:
  + a binary tree whose leaves are the given characters
* Binary codeword for a character is the path from the root to that character in the binary tree
* “” means “**go to the left child**”
* “” means “**go to the right child**”

## Binary Tree Representation of Prefix Codes

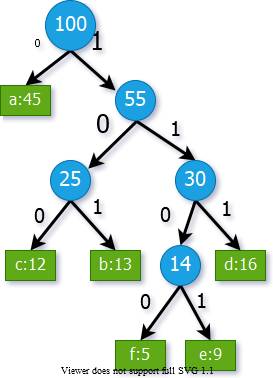
* **Weight of an internal node:** sum of weights of the leaves in its subtree
* The binary tree corresponding to the fixed-length code



center h:450px

## Binary Tree Representation of Prefix Codes

* **Weight of an internal node:** sum of weights of the leaves in its subtree
* The binary tree corresponding to the **optimal variable-length** code
* An optimal code for a file is always represented by a **full binary tree**



bg right:40% h:600px

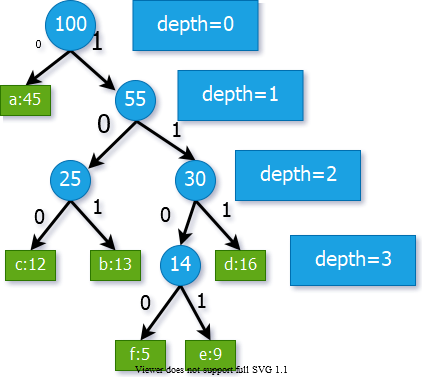
## Full Binary Tree Representation of Prefix Codes

* Consider an **FBT** corresponding to an optimal prefix code
* It has leaves (external nodes)
* One for each letter of the alphabet where is the alphabet from which the characters are drawn
* **Lemma:** An **FBT** with external nodes has exactly internal nodes

## Full Binary Tree Representation of Prefix Codes

* Consider an , corresponding to a prefix code.
* **Notation**:
  + : frequency of character c in the file
  + : depth of ’s leaf in the
  + : the number of bits required to encode the file
* What is the length of the codeword for ?
  + , same as the depth of in
* How to compute , cost of tree ?

## Cost Computation - **Example**

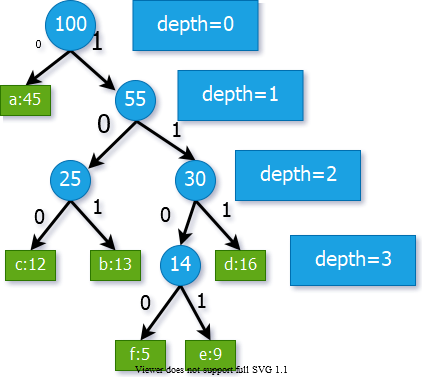


bg right:50% h:500px

## Prefix Codes

* **Lemma:** Let each internal node i is labeled with the sum of the weight of the leaves in its subtree
* Then
* *where is the set of internal nodes of*
* **Proof:** Consider a leaf node with &
  + Then, appears in the weights of internal node
  + along the path from to the root
  + Hence, appears times in the above summation

## Cost Computation - **Example**



bg right:50% h:500px

## Constructing a Huffman Code

* **Problem Formulation:** For a given character set C, construct an optimal prefix code with the minimum total cost
* **Huffman** invented a **greedy algorithm** that constructs an optimal prefix code called a **Huffman code**
* The greedy algorithm
  + builds the **FBT** corresponding to the optimal code in a **bottom-up** manner
  + begins with a set of leaves
  + performs a sequence of “**merges**” to create the final tree

## Constructing a Huffman Code

* A **priority queue** , keyed on , is used to identify the two **least-frequent** objects to merge
* The result of the **merger** of two objects is a **new object**
  + inserted into the priority queue according to its frequency
  + which is the sum of the frequencies of the two objects merged

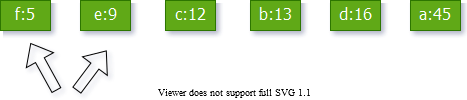
## Constructing a Huffman Code

* Priority queue is implemented as a binary heap
* Initiation of (): time
* & take time on with objects

## Constructing a Huffman Code

## Constructing a Huffman Code - **Example**

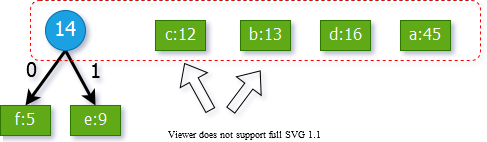
* Start with one leaf node for each character
* The nodes with the least frequencies:
* Merge and create an internal node
* Set the internal node frequency to



center h:300px

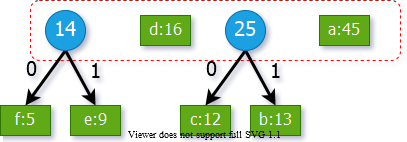
## Constructing a Huffman Code - **Example**

* The 2 nodes with least frequencies:



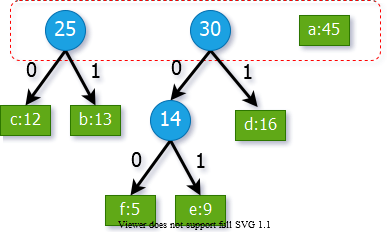
center h:400px

## Constructing a Huffman Code - **Example**



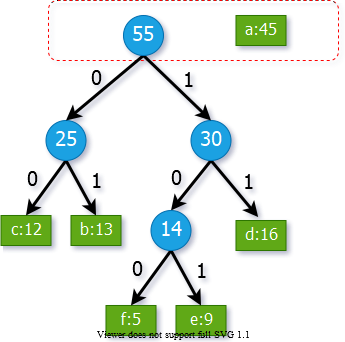
center h:350px

## Constructing a Huffman Code - **Example**



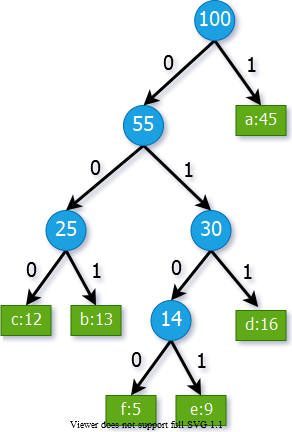
bg right:50% h:350px

## Constructing a Huffman Code - **Example**



bg right:50% h:600px

## Constructing a Huffman Code - **Example**



bg right:50% h:600px

## Correctness Proof of Huffman’s Algorithm

* **We need to prove:**
  + The greedy choice property
  + The optimal substructure property
* **What is the greedy step in Huffman’s algorithm?**
  + *Merging the two characters with the lowest frequencies*
* *We will first prove the greedy choice property*

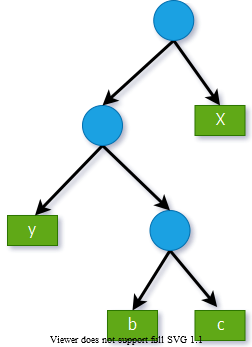
## Greedy Choice Property

* **Lemma 1:** Let be two characters in having the **lowest frequencies**.
* Then, an optimal prefix code for in which the codewords for have the same length and differ only in the last bit
* **Note:** *If are merged in Huffman’s algorithm, their codewords are guaranteed to have the same length and they will differ only in the last bit*.
  + *Lemma 1* states that there exists an optimal solution where this is the case.

## Greedy Choice Property - Proof

* Outline of the proof:
  + Start with an arbitrary optimal solution
  + Convert it to an optimal solution that satisfies the greedy choice property.
* **Proof:** Let be an arbitrary optimal solution where:
  + are the sibling leaves with the **max depth**
  + are the characters with the **lowest frequencies**

## Greedy Choice Property - Proof

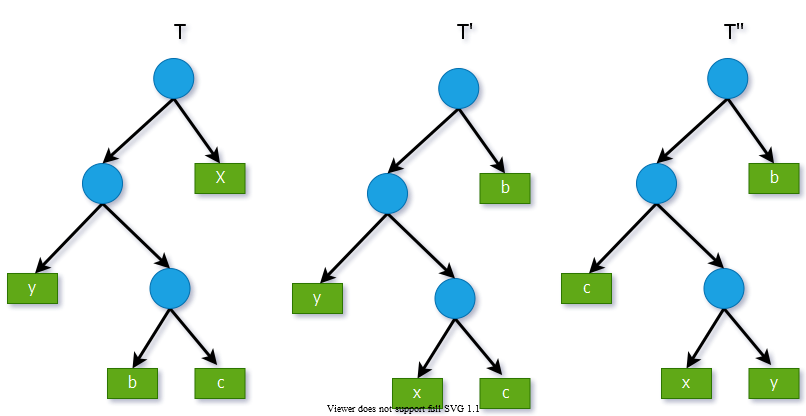


bg right:50% h:600px

* Reminder:
  + are the nodes with max depth
  + are the nodes with min freq.
* Without loss of generality, assume:
* Then, it must be the case that:

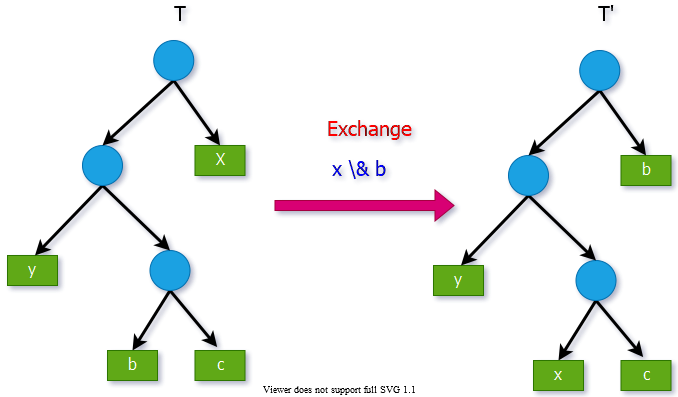
## Greedy Choice Property - Proof

* : exchange the positions of the leaves
* : exchange the positions of the leaves



center h:400px

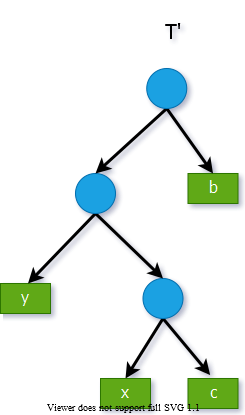
## Greedy Choice Property - Proof



center h:500px

## Greedy Choice Property - Proof

* **Reminder:** Cost of tree
* How does compare to ?
* **Reminder:**
  + and



bg right:50% h:500px

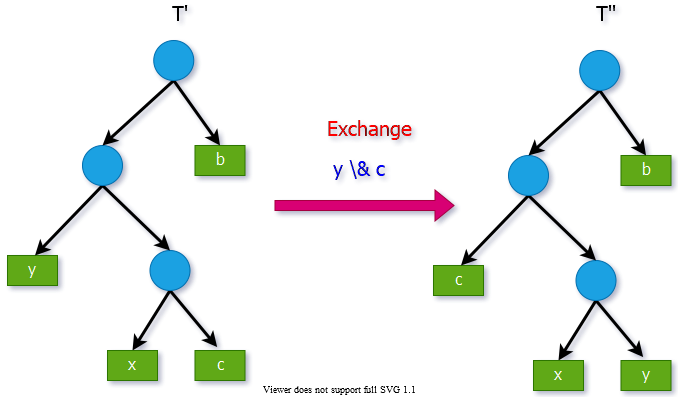
## Greedy Choice Property - Proof

* **Reminder:**
  + and
* The difference in cost between and :

## Greedy Choice Property - Proof

* Since and
  + therefore
* In other words, is also optimal

## Greedy Choice Property - Proof



center h:450px

## Greedy Choice Property - Proof

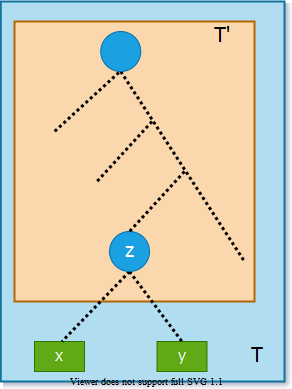
* We can similarly show that
  + which implies
* Since is optimal is also optimal
* **Note:** contains our greedy choice:
  + Characters appear as sibling leaves of max-depth in
* Hence, the proof for the greedy choice property is complete

## Greedy-Choice Property of Determining an Optimal Code

* **Lemma 1** implies that
  + process of building an optimal tree
  + by mergers can begin with the greedy choice of merging
  + those two characters with the lowest frequency
* We have already proved that , that is,
  + the total cost of the tree constructed
  + is the **sum** of the **costs** of its **mergers** (**internal nodes**) **of all possible mergers**
* At each step **Huffman chooses** the merger that incurs the **least cost**

## Optimal Substructure Property

* Consider an optimal solution for alphabet . Let and be any two sibling leaf nodes in . Let be the parent node of and in .
* Consider the subtree where .
  + Here, consider z as a new character, where
* **Optimal substructure property:** must be optimal for the alphabet , where



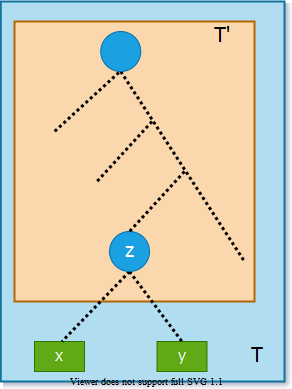
bg right:30% h:500px

## Optimal Substructure Property - Proof

Reminder:

Try to express in terms of .

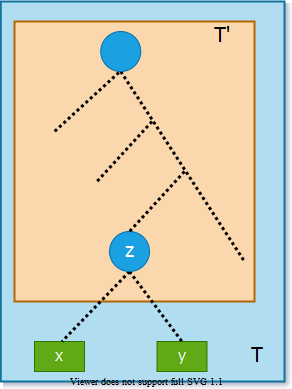
**Note:** All characters in have the same depth in and .



bg right:30% h:500px

## Optimal Substructure Property - Proof

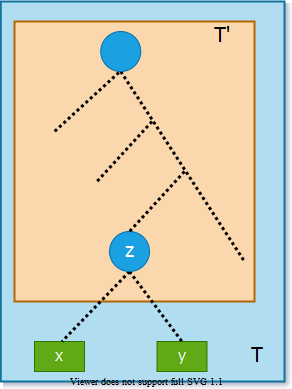
Reminder:



bg right:30% h:500px

## Optimal Substructure Property - Proof

* We want to prove that is optimal for
* Assume by contradiction that that there exists another solution for with smaller cost than . Call this solution :
* Let us construct another prefix tree by adding as children of in

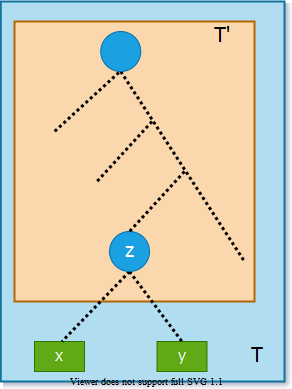


bg right:30% h:500px

## Optimal Substructure Property - Proof

* Let us construct another prefix tree by adding as children of in .
* We have:
* In the beginning, we assumed that:
* So, we have:

**Contradiction! Proof complete**



bg right:30% h:500px

## Greedy Algorithm for Huffman Coding - Summary

* For the greedy algorithm, we have proven that:
  + **The greedy choice property** holds.
  + **The optimal substructure property** holds.
* So, the greedy algorithm is optimal.

## References

* [Introduction to Algorithms, Third Edition | The MIT Press](https://mitpress.mit.edu/books/introduction-algorithms-third-edition)
* [Bilkent CS473 Course Notes (new)](http://nabil.abubaker.bilkent.edu.tr/473/)
* [Bilkent CS473 Course Notes (old)](http://cs.bilkent.edu.tr/~ugur/teaching/cs473/)