Codeword Encoding--Huffman Encoding

Computer Science Department
CS4481b/9628b: Image Compression
Winter 2017
Instructor: Mahmoud R. El-Sakka
Office: MC-419
Email: elsakka@csd.uwo.ca
Phone: 519-661-2111 x86996

Encoding

- Encoding, or simply coding, means the assignment of *binary sequences* to elements of an *alphabet*
- The set of *binary sequences* is called a *code*
- An *individual member of a code* is called *codeword*
- An *alphabet* is a *collection of symbols*
- Example:
  - ASCII is a *fixed-length* 7-bit code
  - In ASCII, the codeword for symbol *a* is $(1100001)_2$
  - In ASCII, the codeword for symbol *A* is $(1000001)_2$

Encoding

- *Fixed-length* codes
  - The simplest way to do encoding
  - Any codeword has the same number of bits
  - The average bit rate per symbol is equal to the codeword length
- If we want to reduce the number of bits required to represent various messages, we need to use a *variable number of bits* to represents individual symbols
  - If we use fewer bits to represent symbols that occur more often, we would use fewer bits per symbol (on the average)

Decoding

- Decoding means the conversion from *codewords* to the correspondence elements of an *alphabet*
- A code is called *uniquely decodable code*, if, and only if, for *any* collection of codewords with any order, you will always have one, and only one, way to decode these codewords

Uniquely Decodable Codes

Example 1: Consider the following code:

$a_1 \rightarrow 0$

$a_2 \rightarrow 0$

$a_3 \rightarrow 1$

$a_4 \rightarrow 10$

- This is not a *uniquely decodable code* because two symbols have assigned the same codeword
- The above code would be unable to transfer information in an unambiguous manner

Uniquely Decodable Codes

Example 2: Consider the following code:

$a_1 \rightarrow 0$

$a_2 \rightarrow 1$

$a_3 \rightarrow 00$

$a_4 \rightarrow 11$

- Each symbol in the above code has a unique codeword
- However, if we want to encode $a_1\, a_1$, we would use *00*, which can be decoded as a_3 as well; hence the above code is not a *uniquely decodable code*

Uniquely Decodable Codes

Example 3: Consider the following code:

$a_1 \rightarrow 0$

$a_2 \rightarrow 10$

$a_3 \rightarrow 110$

$a_4 \rightarrow 111$

- The first three codewords all end in a *0*
- The final codeword contains no *0s* and is a 3 bits long
- This is a *uniquely decodable code*
- The decoding rule for this code is to accumulate bits until
  - you get a *0* or
  - until you have three *1's*

Uniquely Decodable Codes

Example 4: Consider the following code:

$a_1 \rightarrow 0$

$a_2 \rightarrow 01$

$a_3 \rightarrow 011$

$a_4 \rightarrow 0111$

- Each codeword starts with a *0*
- The only time we see a *0* is at the beginning of a codeword
- This is a *uniquely decodable code*
- The decoding rule for this code is even simpler:
  - accumulate bits until you see a *0*
- *Is there any problem with this code?*

Uniquely Decodable Codes

- In example 3, the decoder knows the completion of a codeword from its bits only, i.e, without reading next codeword(s)
- In example 4, the decoder has to wait till reading next codeword(s) to know that the current codeword is completed
- Because of this property, code in example 3 is called *instantaneous code*, while code in example 4 is not
- While it is nice to have the *instantaneous decoding property* in a code, it is not a requirement for a *uniquely decodable code*

Uniquely Decodable Codes

Example 5: Consider the following code:

$a_1 \rightarrow 0$

$a_2 \rightarrow 01$

$a_3 \rightarrow 11$

- Is this code a *uniquely decodable code*?
- Is this code an *instantaneous code*?
- What is the difference between codes in examples 4 and 5?

Example 4:

$a_1 \rightarrow 0$

$a_2 \rightarrow 01$

$a_3 \rightarrow 011$

$a_4 \rightarrow 0111$

Uniquely Decodable Codes

Example 6: Consider the following code:

$a_1 \rightarrow 0$

$a_2 \rightarrow 01$

$a_3 \rightarrow 10$

Example 5:
$a_1 \rightarrow 0$
$a_2 \rightarrow 01$
$a_3 \rightarrow 11$

- Is this code a *uniquely decodable code*?
- Is this code an *instantaneous code*?
- What is the difference between codes in examples 5 and 6?

Uniquely Decodable Codes

- So far, we looked at small codes with three or four symbols
- Even with these, it is not immediately evident whether the code is uniquely decodable code or not
- Hence, a systematic procedure to test for unique decodability would be useful

Uniquely Decodable Codes

- Definitions:
  Suppose we have two binary codewords a and b, where
  - a is k bits long
  - b is n bits long
  - $k < n$

  If the first k bits in b are identical to a, then
  - a is called a *prefix* of b
  - The last $n - k$ bits of b are called the *dangling suffix*

- Example:
  if a = 010 and b = 01011
  - a is a *prefix* of b
  - The *dangling suffix* is 11

Uniquely Decodable Codes

Procedure to test for unique decodability

- Examine all pairs of codewords
  - Whenever you find a codeword is a *prefix* of another codeword
    - If the *dangling suffix* is equal to any of the original codeword,
      Then
        The code is ***NOT uniquely decodable code***
      Else
        Add the *dangling suffix* to the *augmented codeword list*, ***unless*** you have added the same dangling suffix to the list in a previous iteration
- Repeat the above step, but this time between ***each codeword*** and each ***dangling suffix*** **(but not between two *dangling suffixs*)** until one of the following two things occurs
  - You ***get*** a *dangling suffix* that is a ***codeword***
    (the code is ***NOT uniquely decodable code***)
  - There are ***no more unique*** *dangling suffixes* to be added to the list
    (the code is ***uniquely decodable code***)

Uniquely Decodable Codes

Example 7: Consider the code in example 5: $a_1 \rightarrow 0$ $a_2 \rightarrow 01$ $a_3 \rightarrow 11$, *use dangling suffix* and *augmented codeword list* to determine if this code is *uniquely decodable* or not.

- The first list of codewords is {0, 01, 11}
- Comparing the elements of this list, we found
  - 0 is a prefix for 01 with a dangling suffix of 1
  - The dangling suffix 1 is not a codeword and we never added it to the list in a previous iteration, so we add it to the list
  - Now the augmented list is {0, 01, 11, **1**}
- Comparing the elements of this augmented list,
  - 1 is a prefix for 11 with a dangling suffix of 1
  - The dangling suffix 1 is not a codeword ***BUT*** we have added it to the list in a previous iteration, so we will not add it again
- There are *no more unique dangling suffixes* to be added to the list and we *got no dangling suffix that is a codeword*, ***hence the code is a uniquely decodable code***

Topic 03: Codeword Encoding--Huffman Encoding

Uniquely Decodable Codes

Example 8: Consider the code in example 6: $a_1 \rightarrow 0$ $a_2 \rightarrow 01$ $a_3 \rightarrow 10$, *use dangling suffix* and *augmented codeword list* to determine if this code is *uniquely decodable* or not.

- The first list of codewords is {0, 01, 10}
- Comparing the elements of this list, we found
  - 0 is a prefix for 01 with a dangling suffix of 1
  - The dangling suffix 1 is not a codeword and we never added it to the list in a previous iteration, so we add it to the list
  - Now the augmented list is {0, 01, 10, **1**}
- Comparing the elements of this augmented list,
  - 1 is a prefix for the codeword 10 with a dangling suffix of 0
  - The dangling suffix 0 is the codeword for a_1
- Since we *got a dangling suffix that is a codeword*, ***the code is NOT a uniquely decodable code***

Uniquely Decodable Codes

Example 8: Consider the code in example 6: $a_1 \rightarrow 0$ $a_2 \rightarrow 01$ $a_3 \rightarrow 10$, *use dangling suffix* and *augmented codeword list* to determine if this code is *uniquely decodable* or not.

- To generate a counter example, you trace prefix cases from the original codewords till finding a dangling suffix that is a codeword as well.
  - 0 is a prefix for 01 with a dangling suffix of 1
  - 1 is a prefix for the codeword 10 with a dangling suffix of 0
  - The dangling suffix 0 is the codeword for a_1
- The counter example is the concatenation of the codewords 01 and 10 *after removing the overlap that occurs due to the dangling suffix*, i.e., the 1 in this case
  - `01`
  - ` 10`
  - `010` ← `The is the counter example`
    - Can be decoded as $a_2\ a_1$
    - Can be decoded as $a_1\ a_3$

Uniquely Decodable Codes

Example 9: Consider the following code :
$a_1 \rightarrow 101$ $a_2 \rightarrow 1011$ $a_3 \rightarrow 0100$ $a_4 \rightarrow 00101$,
use dangling suffix and *augmented codeword list* to determine if this code is *uniquely decodable* or not.

- **a_1 and a_2 → producing r_1 = 1**
- **r_1 and a_1 → producing r_2 = 01**
- r_1 and a_2 → producing r_3 = 011
- **r_2 and a_3 → producing r_4 = 00**
- **r_4 and a_4 → producing r_5 = 101 ← also a_1**
- Since we *got a dangling suffix that is a codeword*, ***the code is NOT a uniquely decodable code***
- The counter example (go backward from the last prefix to the first one, i.e, r_5 line, r_4 line, r_2 line , r_1 line (*note that we do not used , r_3 line*)
  - From r_1 line: `101` ← a_1
  - `1011` ← a_2
  - From r_2 line: `101` ← a_1
  - From r_4 line: `0100` ← a_3
  - From r_5 line: `00101` ← a_4
  - `10110100101` ← `The is the counter example`
    - Can be decoded as $a_1\ a_1\ a_4$ ← `10110100101`
    - Can be decoded as $a_2\ a_3\ a_1$ ← `10110100101`

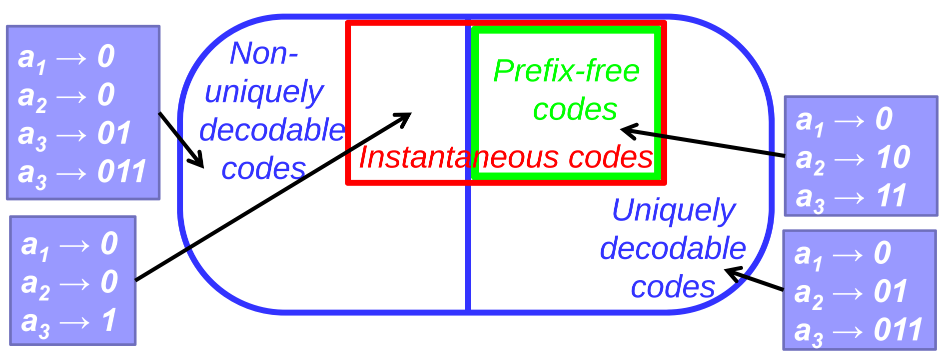
Uniquely Decodable Codes

- ***NB1***: this method does not test whether the code is *instantaneous* or not

- **NB2**: For a give code, if ***no*** codeword is a prefix of the other, the set of dangling suffixes will always be empty and hence we do not have to worry about a dangling suffix that is identical to a codeword
  - Such code is called *prefix-free code* (or simply *prefix code*)
  - A *prefix-free code* is *uniquely decodable code* as well as *instantaneous code*
  - *uniquely decodable codes* are *not necessarily prefix-free codes,* see example 7

Codes summary

- Codes can be
  - *uniquely decodable codes* or not
  - *instantaneous codes* or not
  - *Prefix-free codes* or not



Huffman Encoding

- Huffman encoding scheme is more than half century old; it was proposed in 1951
- It is one of the most popular technique for removing encoding redundancy
- In Huffman, symbols with higher probabilities to occur are assigned shorter *prefix-free codes* (consequently the generated code will be *uniquely decodable codes* as well as *instantaneous code*)

Topic 03: Codeword Encoding--Huffman Encoding

Huffman Encoding

- The procedure for building the Huffman code is simple
  - The individual symbols are laid out as an array of unmarked nodes that are going to be connected by a binary tree
  - Each of these nodes has a weight, which is simply the frequency, or the probability, of the symbol's appearance
  - While there are more than one unmarked node, repeat the following steps
    - Locate two unmarked nodes with the lowest weights
    - Create an unmarked parent node for these two nodes
    - assign to this parent node a weight equal to the sum of the two child nodes
    - Arbitrarily label the two parent/child paths with "0" and "1", one label per path
    - Mark the two child nodes
  - The code for each symbol is the accumulated labels
    - starting from *the root* (i.e., the only unmarked node) and
    - ending at each symbol (*at the leaf*)

Huffman Encoding

Example 1:

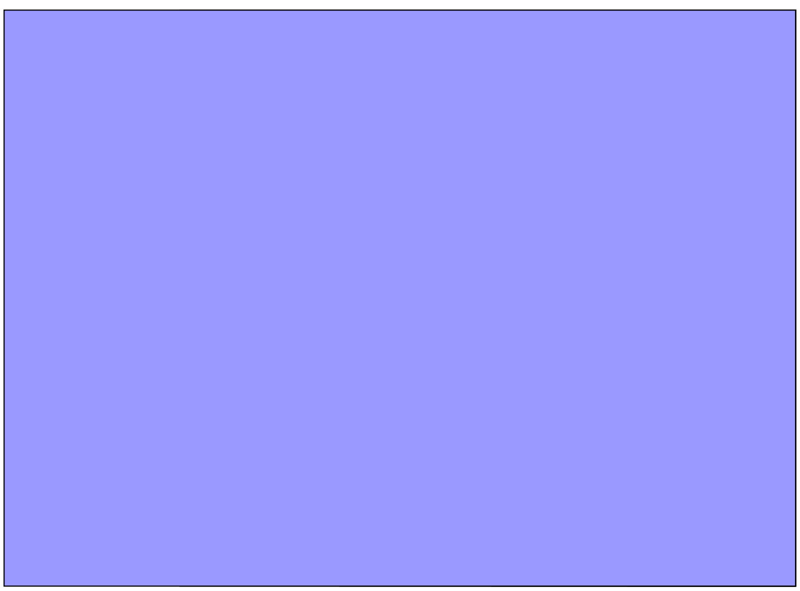
- Consider a set of symbols R = $\{r_1, r_2, r_3, r_4, r_5, r_6\}$
- The probability of occurrence of these symbols are:
  - $P(r_1) = 20/32$
  - $P(r_2) = 3/32$
  - $P(r_3) = 3/32$
  - $P(r_4) = 1/32$
  - $P(r_5) = 4/32$
  - $P(r_6) = 1/32$

  Design a Huffman code for these symbols
  and
  calculate the average bit rate per each symbol.

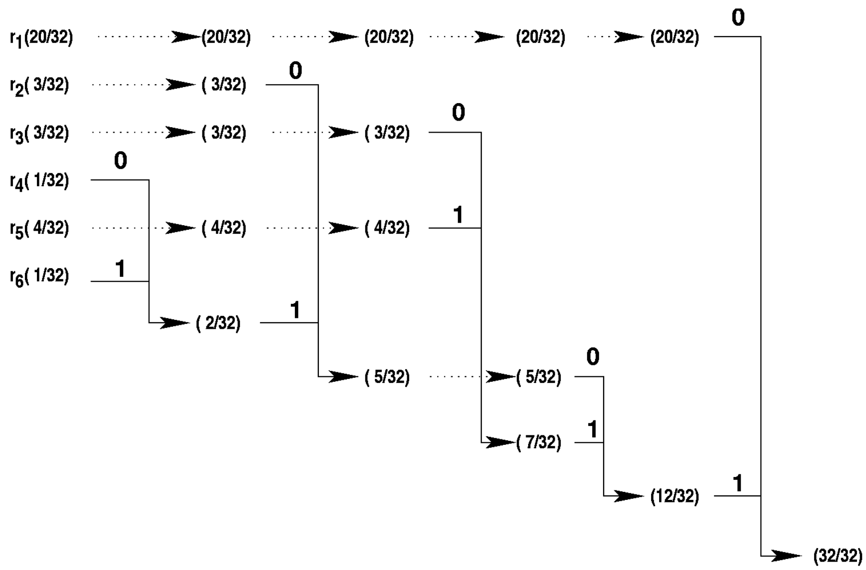
Topic 03: Codeword Encoding--Huffman Encoding

Huffman Encoding

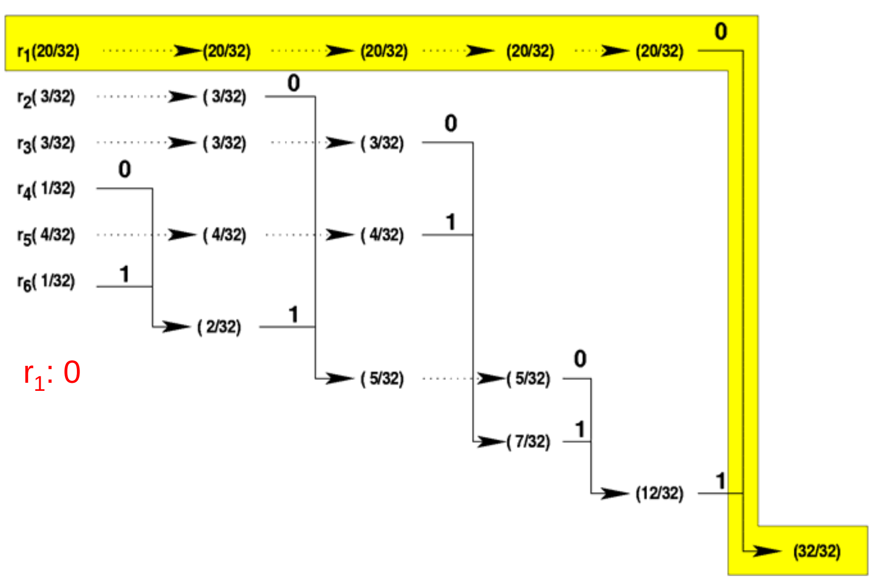
r_1(20/32)
r_2(3/32)
r_3(3/32)
r_4(1/32)
r_5(4/32)
r_6(1/32)



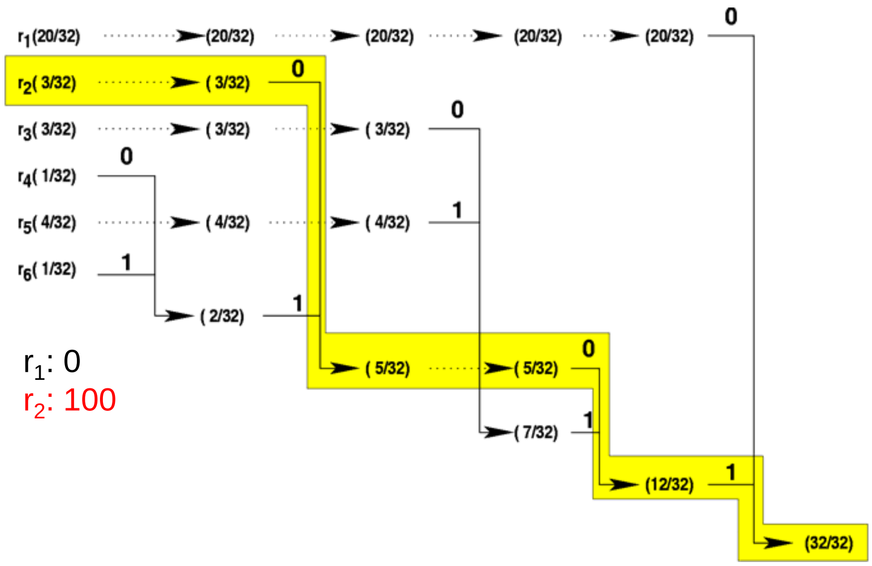
Huffman Encoding



Huffman Encoding

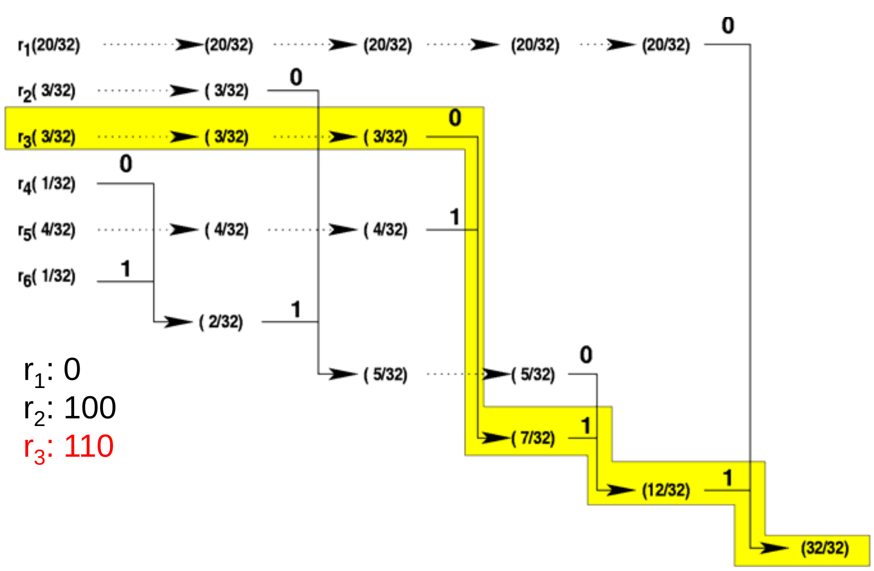


Huffman Encoding



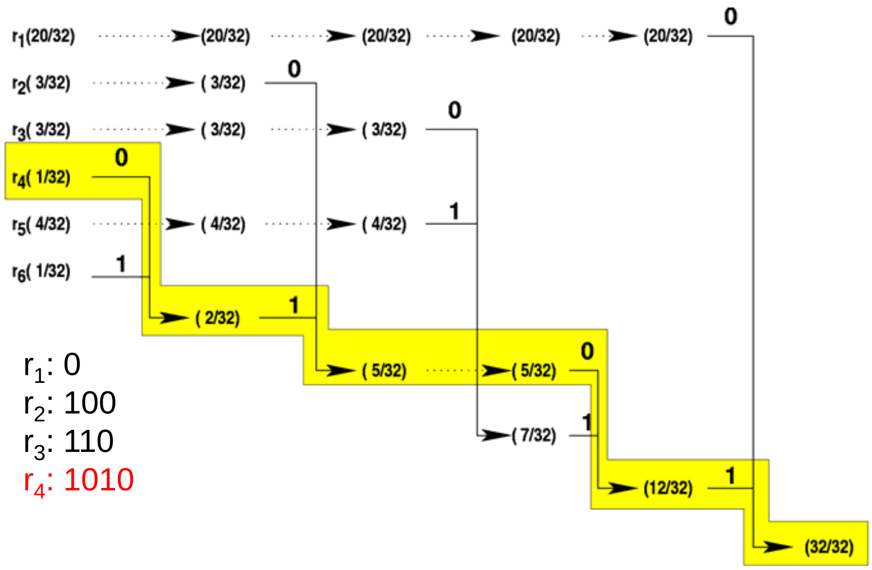
r_1: 0
r_2: 100

Huffman Encoding



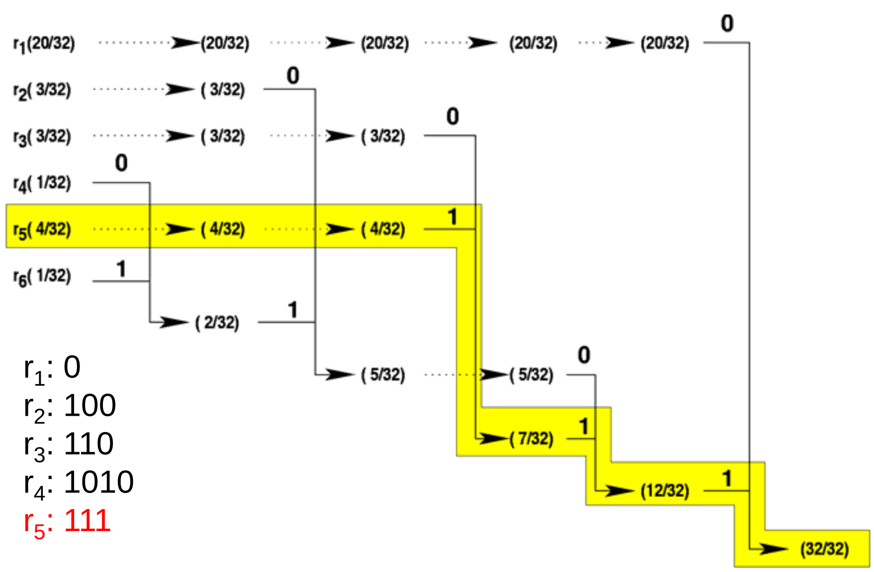
r_1: 0
r_2: 100
r_3: 110

Huffman Encoding



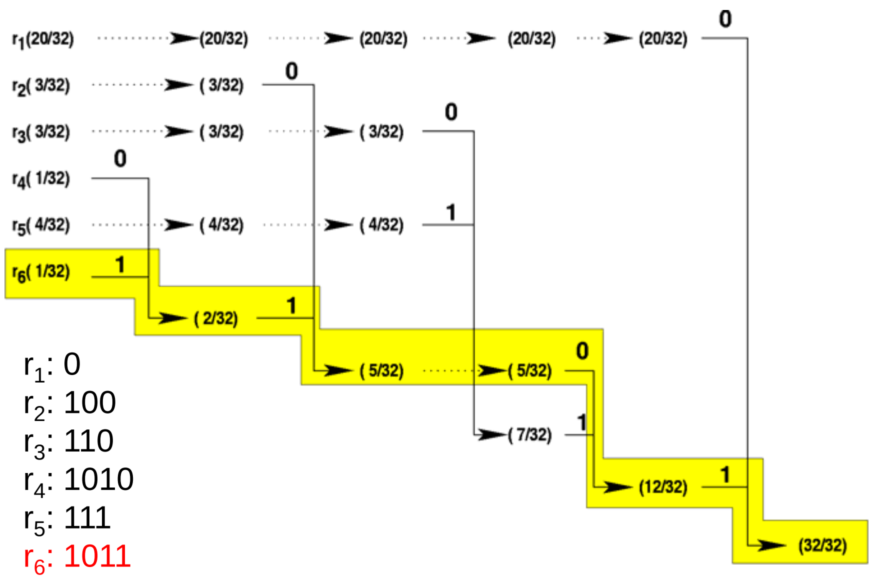
r_1: 0
r_2: 100
r_3: 110
r_4: 1010

Huffman Encoding



r_1: 0
r_2: 100
r_3: 110
r_4: 1010
r_5: 111

Huffman Encoding



r_1: 0
r_2: 100
r_3: 110
r_4: 1010
r_5: 111
r_6: 1011

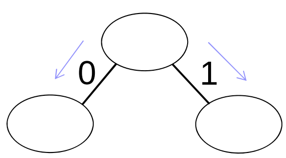
Huffman Encoding

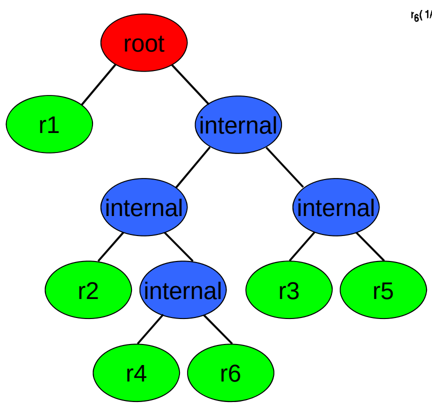
Symbol	Codeword	Codeword length	Symbol probability	Expected codeword length
r_1	0	1 bit	20/32	20/32 bits
r_2	100	3 bits	3/32	9/32 bits
r_3	110	3 bits	3/32	9/32 bits
r_4	1010	4 bits	1/32	4/32 bits
r_5	111	3 bits	4/32	12/32 bits
r_6	1011	4 bits	1/32	4/32 bits
			Average bit rate =	58/32 = 1.8125 bits/symbol

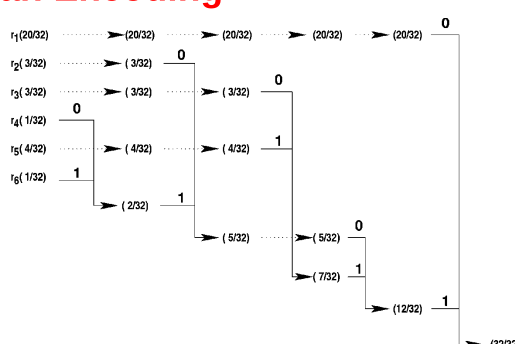
Huffman Encoding

- How to draw a Huffman tree?
  - Start from a single node (the root node)
  - Each node can have either
    - zero branch (in this case we call it external node, or leave node)
    - Two branches (in this case we call it internal node)
      - One of these branches corresponds to a 0 (assume it is the left branch)
      - The other branch corresponds to 1 (assume it is the right branch)

Huffman Encoding







r_1: 0
r_2: 100
r_3: 110
r_4: 1010
r_5: 111
r_6: 1011

How to Generate Huffman Codes or Tree

Using probabilities (floating or rational numbers)

r_1	20/32	r_1	20/32	r_1	20/32	r_1	20/32	r_1	**20/32**	r_1	32/32
r_2	3/32	r_2	**3/32**	r_2	5/32	r_2	**5/32**	r_2	**12/32**	r_2	
r_3	3/32	r_3	3/32	r_3	**3/32**	r_3	**7/32**	r_3		r_3	
r_4	**1/32**	r_4	**2/32**	r_4		r_4		r_4		r_4	
r_5	4/32	r_5	4/32	r_5	**4/32**	r_5		r_5		r_5	
r_6	**1/32**	r_6		r_6		r_6		r_6		r_6	

(4,6) **(2,4)** **(3,5)** **(2,3)** **(1,2)**

How to Generate Huffman Codes or Tree

Using frequencies (integer numbers)

r_1	20	r_1	20	r_1	20	r_1	20	r_1	**20**	r_1	32
r_2	3	r_2	**3**	r_2	5	r_2	**5**	r_2	**12**	r_2	
r_3	3	r_3	3	r_3	**3**	r_3	**7**	r_3		r_3	
r_4	**1**	r_4	**2**	r_4		r_4		r_4		r_4	
r_5	4	r_5	4	r_5	**4**	r_5		r_5		r_5	
r_6	**1**	r_6		r_6		r_6		r_6		r_6	

(4,6) **(2,4)** **(3,5)** **(2,3)** **(1,2)**

How to Generate Huffman Codes or Tree

- To generate Huffman codes, or a Huffman tree, you only need the locations of the join pairs in backward order; i.e,
  - (1,2)
  - (2,3)
  - (3,5)
  - (2,4)
  - (4,6)
- The number of these pairs equal to

  the total number of symbols – 1

How to Generate Huffman Codes

r_1=0

r_2=1

r_3=

r_4=

r_5=

r_6=

(4,6) (2,4) (3,5) (2,3) **(1,2)**

How to Generate Huffman Codes

r_1=0

r_2=10

r_3=11

r_4=

r_5=

r_6=

(4,6) (2,4) (3,5) **(2,3)** (1,2)

Topic 03: Codeword Encoding--Huffman Encoding

How to Generate Huffman Codes

r_1=0

r_2=10

r_3=110

r_4=

r_5=111

r_6=

(4,6) (2,4) **(3,5)** (2,3) (1,2)

How to Generate Huffman Codes

r_1=0

r_2=100

r_3=110

r_4=101

r_5=111

r_6=

(4,6) **(2,4)** (3,5) (2,3) (1,2)

How to Generate Huffman Codes

r_1=0

r_2=100

r_3=110

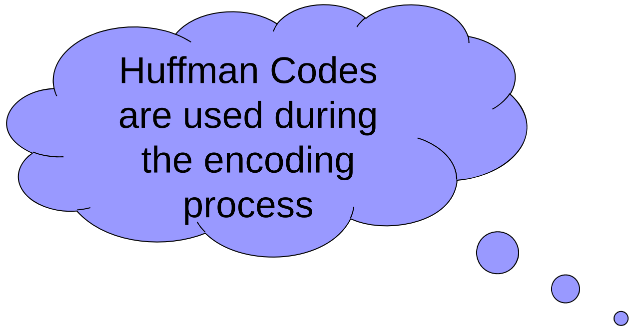
r_4=1010

r_5=111

r_6=1011

(4,6) (2,4) (3,5) (2,3) (1,2)

How to Generate Huffman Codes



r_1=0

r_2=100

r_3=110

r_4=1010

r_5=111

r_6=1011

(4,6) (2,4) (3,5) (2,3) (1,2)

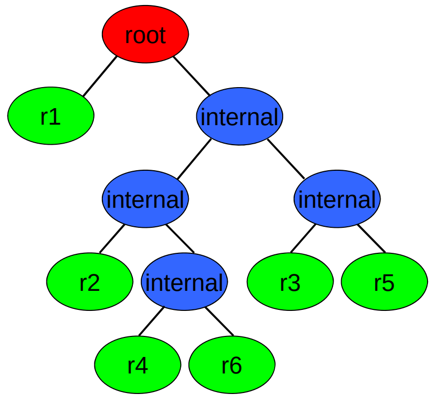
Huffman Encoding Procedure

- While the end-of-symbols-need-to-be-encoded is not reached
  - Read a single symbol from the input data
  - Find the corresponding Huffman code for that symbol
  - Concatenate this Huffman code to the compressed Huffman data

- Example:
  - $r_5\ r_1\ r_1\ r_5\ r_2\ r_6\ r_3\ r_1\ r_1\ r_4$ will be encode as 1110011110010111100 01010

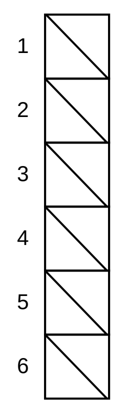
r_1: 0
r_2: 100
r_3: 110
r_4: 1010
r_5: 111
r_6: 1011

How to Generate a Huffman Tree



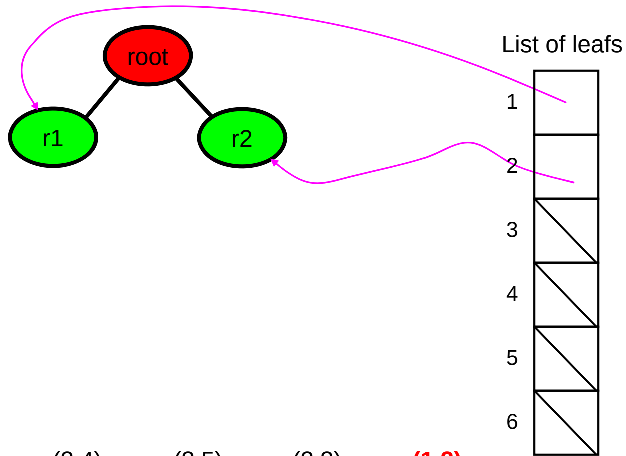
How to Generate a Huffman Tree

List of leafs



(4,6) (2,4) (3,5) (2,3) **(1,2)**

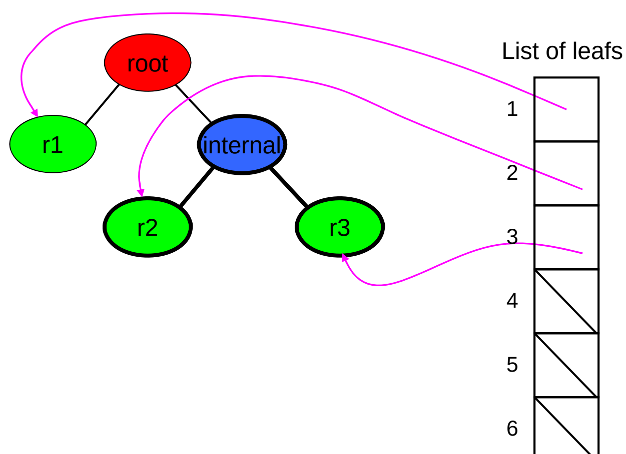
How to Generate a Huffman Tree



(4,6) (2,4) (3,5) (2,3) **(1,2)**

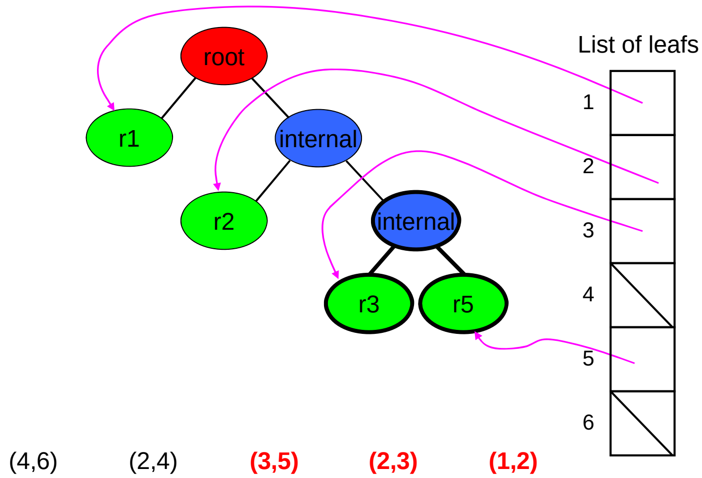
Topic 03: Codeword Encoding--Huffman Encoding

How to Generate a Huffman Tree

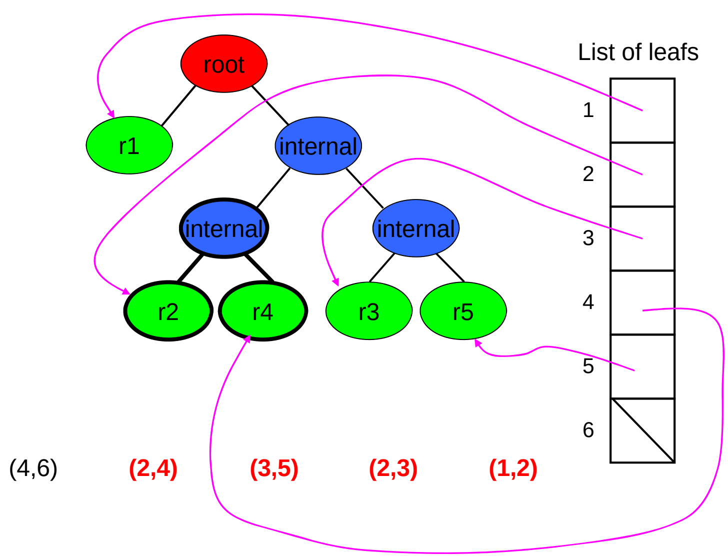


(4,6) (2,4) (3,5) **(2,3)** **(1,2)**

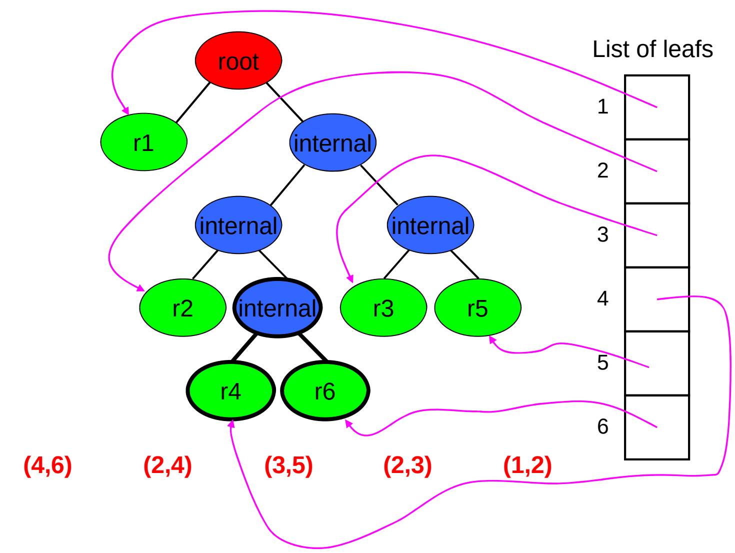
How to Generate a Huffman Tree



How to Generate a Huffman Tree

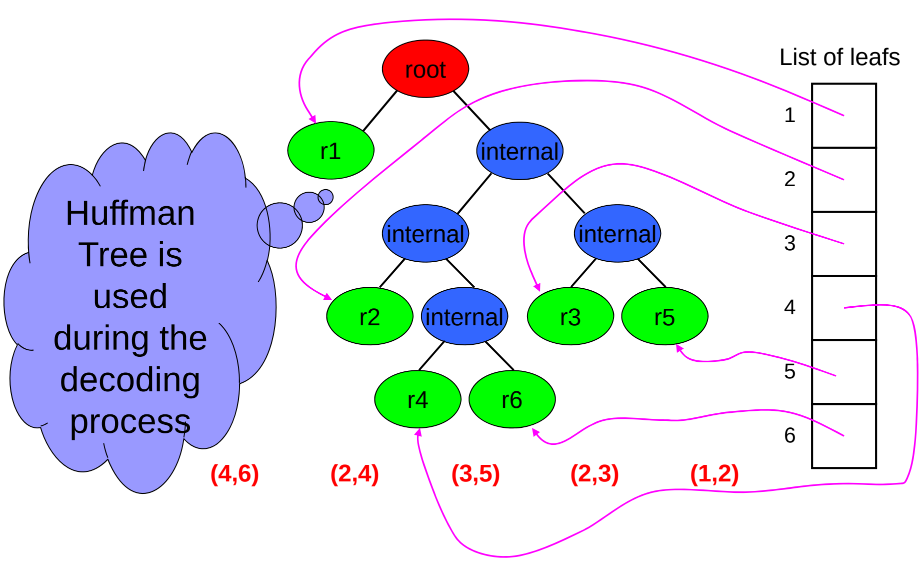


How to Generate a Huffman Tree



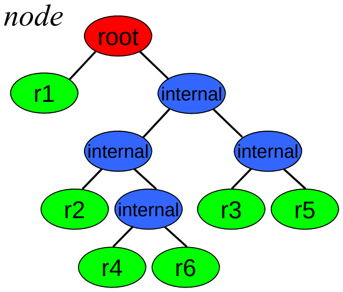
Topic 03: Codeword Encoding--Huffman Encoding

How to Generate a Huffman Tree



Huffman Decoding Procedure

- While the end-of-compressed-Huffman-data is not reached
  - *current node* = root node
  - While *current node* is not a leaf node
    - Read a single bit from the compressed Huffman data
      - If it is 0, go to the left child node and declare it as the *current node*
      - If it is 1, go to the right child node and declare it as the *current node*
  - Get the decoded symbol from the *current node*
- Example:
  - 111001111001011110001010
    will be encode as
    $r_5\ r_1\ r_1\ r_5\ r_2\ r_6\ r_3\ r_1\ r_1\ r_4$



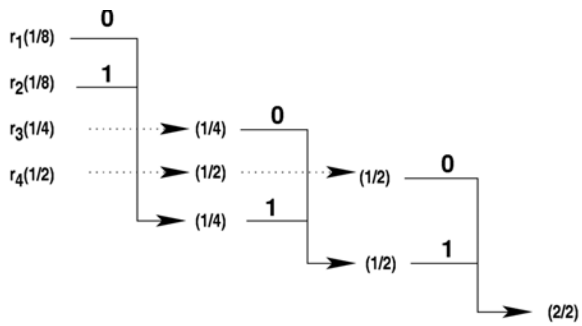
Huffman Encoding

Example 2:

- Consider a set of symbols R = $\{r_1, r_2, r_3, r_4\}$
- The probability of occurrence of these symbols are:
  - $P(r_1) = 1/8$
  - $P(r_2) = 1/8$
  - $P(r_3) = 1/4$
  - $P(r_4) = 1/2$

  Design a Huffman code for these symbols and calculate the average bit rate per each symbol.

Huffman Encoding

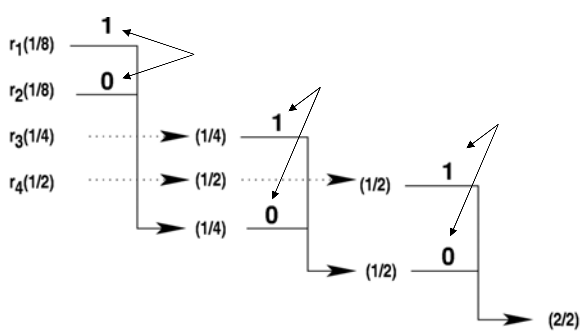


r_1: 110
r_2: 111
r_3: 10
r_4: 0

Average bit rate = 3 × 1/8 + 3 × 1/8 + 2 × 1/4 + 1 × 1/2 = 1.75 bits per symbol

What will happen if I flip the “0” with the “1” in the above figure?

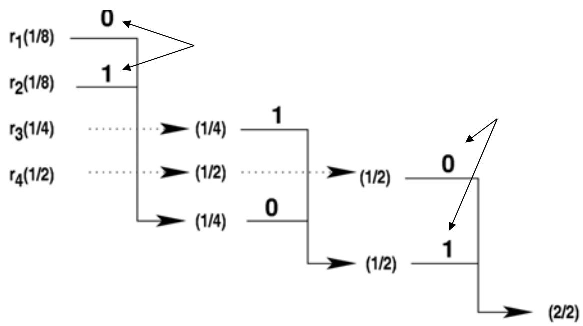
Huffman Encoding



r_1: 001
r_2: 000
r_3: 01
r_4: 1

Average bit rate = 3 × 1/8 + 3 × 1/8 + 2 × 1/4 + 1 × 1/2 = 1.75 bits per symbol

Huffman Encoding



r_1: 100
r_2: 101
r_3: 11
r_4: 0

Average bit rate = 3 × 1/8 + 3 × 1/8 + 2 × 1/4 + 1 × 1/2 = 1.75 bits per symbol

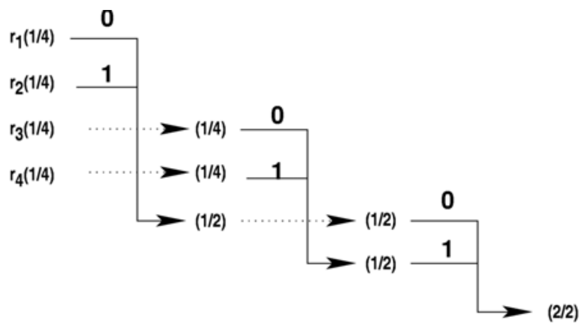
Huffman Encoding

Example 3:

- Consider a set of symbols R = {r_1, r_2, r_3, r_4}
- The probability of occurrence of these symbols are:
  - $P(r_1) = 1/4$
  - $P(r_2) = 1/4$
  - $P(r_3) = 1/4$
  - $P(r_4) = 1/4$

Design a Huffman code for these symbols and calculate the average bit rate per each symbol.

Huffman Encoding



r_1: 00
r_2: 01
r_3: 10
r_4: 11

Average bit rate = 2 × ¼ + 2 × ¼ + 2 × ¼ + 2 × ¼ = 2 bits per symbol