Special course in Computer Science: Advanced Text Algorithms

Lecture 11: Compression algorithms

Eugen Czeizler
Department of IT, Abo Akademi
http://combio.abo.fi/teaching/textalg/

(slides originally by I. Petre, E. Czeizler, V. Rogojin)

Data compression

- Data compression:
  - Reduce the size of the representation of a source of information (e.g. a data file, an image, or a video signal), while preserving the original content as much as possible.
  - Compressed data can only be understood if the decoding method is known by the receiver.
- The goals of data compression algorithms are very practical:
  - Reduce the memory space necessary to store the information.
  - Accelerate data transmission in telecommunication: the smaller the file to be transmitted the faster it can be transferred.

Data compression

- Data compression has a wide range of applications
  - we can say that data compression is used almost everywhere.

- Common image compression formats: JPEG, JPEG 2000, BMP, GIF, PCX, PNG, TGA, TIFF, WMP

- Common audio (sound) compression formats: MPEG-1 Layer III (known as MP3), RealAudio (RA, RAM, RP), AU, Vorbis, WMA, AIFF, WAVE

- Common video (sound and image) compression formats: MPEG-1, MPEG-2, MPEG-4, DivX, Quicktime (MOV), RealVideo (RM), Windows Media Video (WMV), Video for Windows (AVI), Flash video (FLV)

Types of data compression

- Data compression techniques are broadly classified into 2 types: lossy and lossless .
- Lossy data compression techniques reduce the size of a file by permanently eliminating certain redundant information.
- Lossless data compression techniques enable exact reconstruction of the original document from the compressed information.

Lossy data compression

- Lossy techniques are much more effective at compression than lossless methods.
- The original message can never be recovered exactly as it was before it was compressed.
  - Thus, it is not suitable for critical data, when we cannot afford loosing even a single bit.
- These methods are used mostly in sound, video, image compressions where the losses can be tolerated.
  - Examples: JPG, MPEG
- They use a threshold level for the truncation of information.
  - for example, in a sound file, very high and very low frequencies, which cannot be sensed by the human ear, may be eliminated from the file.

Lossles data compression

- The original message can be recovered completely without losing any information.
- These methods exploit the existence of redundancy in data, i.e., repeated patterns in a message are found and encoded in an efficient manner.
- They are most often used for textual data, executable code, word processing files, tabulated numbers.
  - Popular algorithms: Lempel-Ziv, Run Length Encoding, Huffman coding, Arithmetic Coding, Delta Encoding.
- These methods can be also used for image compression: GIF images

Huffman Coding

- Proposed by David A. Huffman in 1952, while he was a PhD student at MIT
  - Adopted in various applications such as fax, JPEG, MP3
- This method is a form of statistical coding that associates to each character in a data file a binary code.
- For different letters, the associated codes can be of different lengths:
  - the most frequent characters in the input file (characters with higher probability of occurring) are associated with short binary codes
  - the least common characters (with lower occurrence probabilities) are associated with longer binary codes.

Associating codes to letters

- It is easy to assign bit-strings to letters.
- Example: A→0, B →01, C →00
  - Then, the text ABC is coded as: 00100.
  - However, the code 010000 corresponds to several words: BAAAA, BCC, BCAA, BACA, BAAC
- Problem: How do we ensure a unique decoding?
- Solution: We should use a prefix code, i.e., a set of words such that no word is a prefix of another one.
- Example: A → 0, B → 100, C → 1010, D → 1011, R → 11
  - The code 0100110101001011010011 0 encodes the word ABRACADABRA

Prefix codes

- Thus, by associating to each letter a bit-string from a prefix code we insure a unique decoding.
- Question: Is the power of the Huffman coding technique decreased if we consider prefix codes?
- Answear: No!
- Theorem: A uniquely decipherable code with given word lengths exists if and only if a prefix code with the same word lengths exists.

Huffman Coding

- This method computes a prefix code C according to a given distribution of frequencies of the letters.

- The entire Huffman compression algorithm has 3 steps:

1. The number of occurrences of each letter is computed.
   - Let n_a be the number of occurrences of letter a in the given text
2. The set $\{n_a \mid a \in \Sigma\}$ is used to compute a prefix code C.
3. Finally, the text is encoded with the prefix code C found at step 2.

Huffman coding

- The prefix code must be sent with the compressed information to enable the receiver decode the message.
  - It is commonly put inside a header of the compressed file.

- For natural language texts, instead of computing the number of occurrences of each letter, the code can be computed using the known letter frequencies:
  - then only step 3 of the algorithm is executed.
  - in this case, the coding is no longer optimal for the given text.

Huffman coding

- The core of the Huffman algorithm is Step 2: the construction of the prefix code, over the alphabet {0,1}, corresponding to the distribution of letters $\{n_a \mid a \in \Sigma\}$.

- It actually constructs a complete binary tree for the prefix code:
  - The prefix condition of the code ensures the fact that there is a one-to-one correspondence between the codewords and the leaves of the tree.

Step 2: The construction of the tree

- For each distinct character a, create a one-node tree containing the character a, and having priority equal with its frequency n_a;
- Create a priority queue of the one-node trees in increasing order of frequency;
- while (there are at least two trees in the priority queue) do begin
  Dequeue the first two trees t_1 and t_2;
  Create a tree t that contains t_1 as its left subtree and t_2 as its right subtree;
  priority(t) = priority(t_1) + priority(t_2);
  insert t in its proper location in the priority queue;
  end
- Assign 0 and 1 weights to the edges of the resulting tree, such that the left and right edge of each node do not have the same weight
  - For instance, for each node assign 0 to the left outgoing edge and 1 to the right outgoing edge.

Example for the construction of the tree

- Suppose we have a text containing only characters a, e, l, n, o, t, and space having the following frequencies:

Char.	Freq.	Char.	Freq.
a	n_a=45	o	n_o=18
e	n_e=65	t	n_t=53
l	n_l=13	space	n_{sp}=22
n	n_n=45		

Example for the construction of the tree (2)

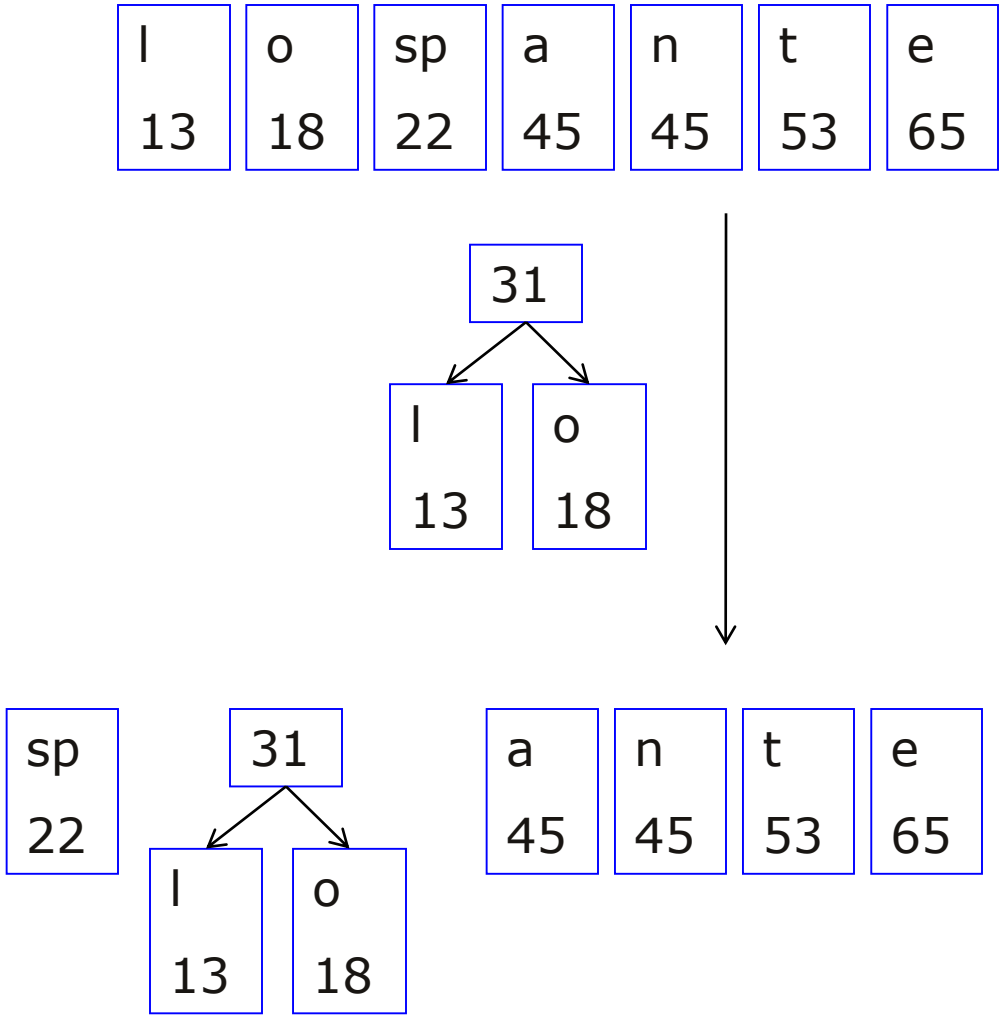
- First, we create a one-node tree for each character, having its frequency n_a as its priority
- Then, we place these one-node trees in a priority queue in increasing order of frequency

l	o	sp	a	n	t	e
13	18	22	45	45	53	65

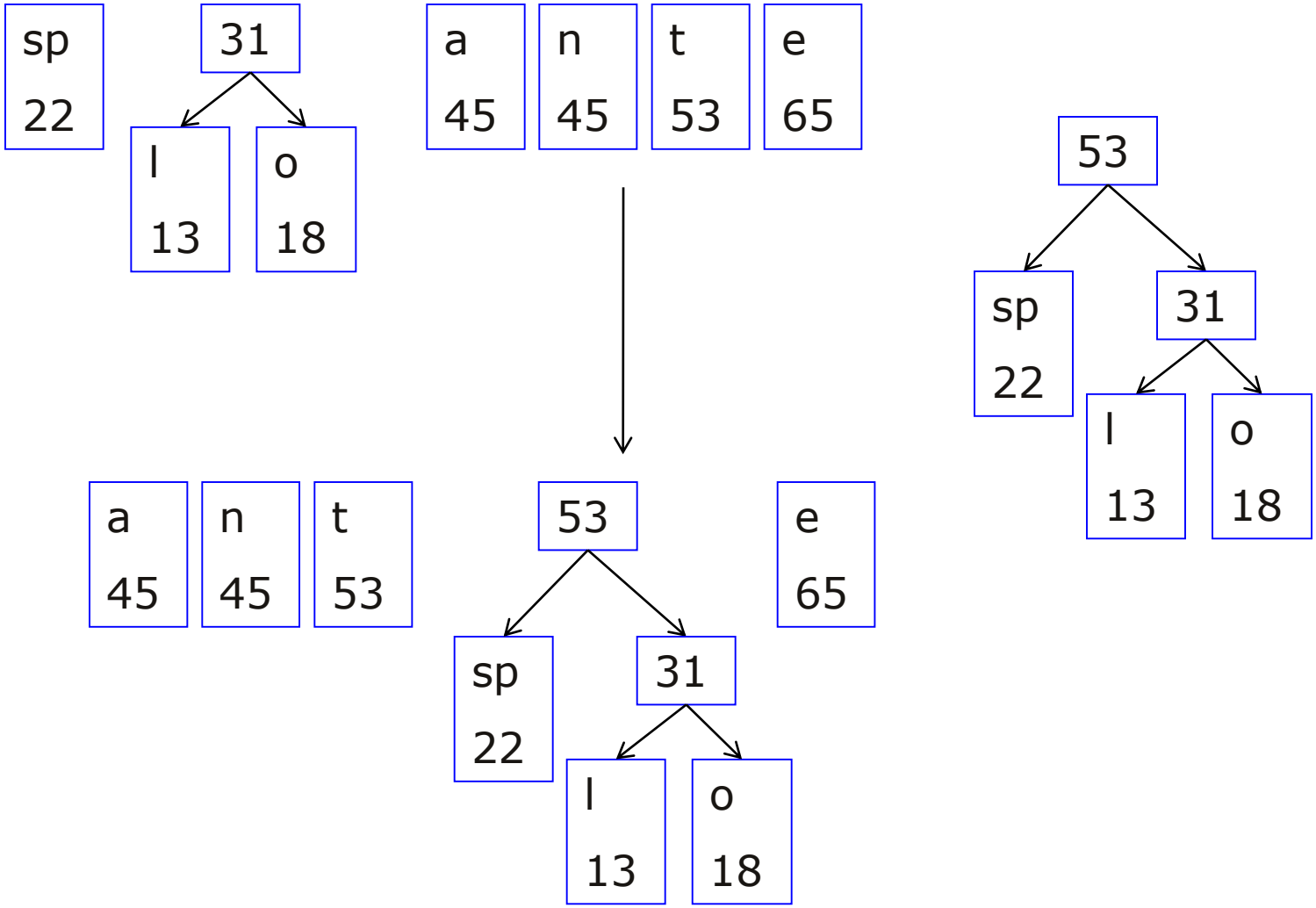
- Select the first 2 trees t_1 and t_2 and crate a new tree t that contains t_1 as its left subtree and t_2 as its right subtree; having priority(t) = priority(t_1) + priority(t_2);

- Then, insert t in its proper location in the priority queue

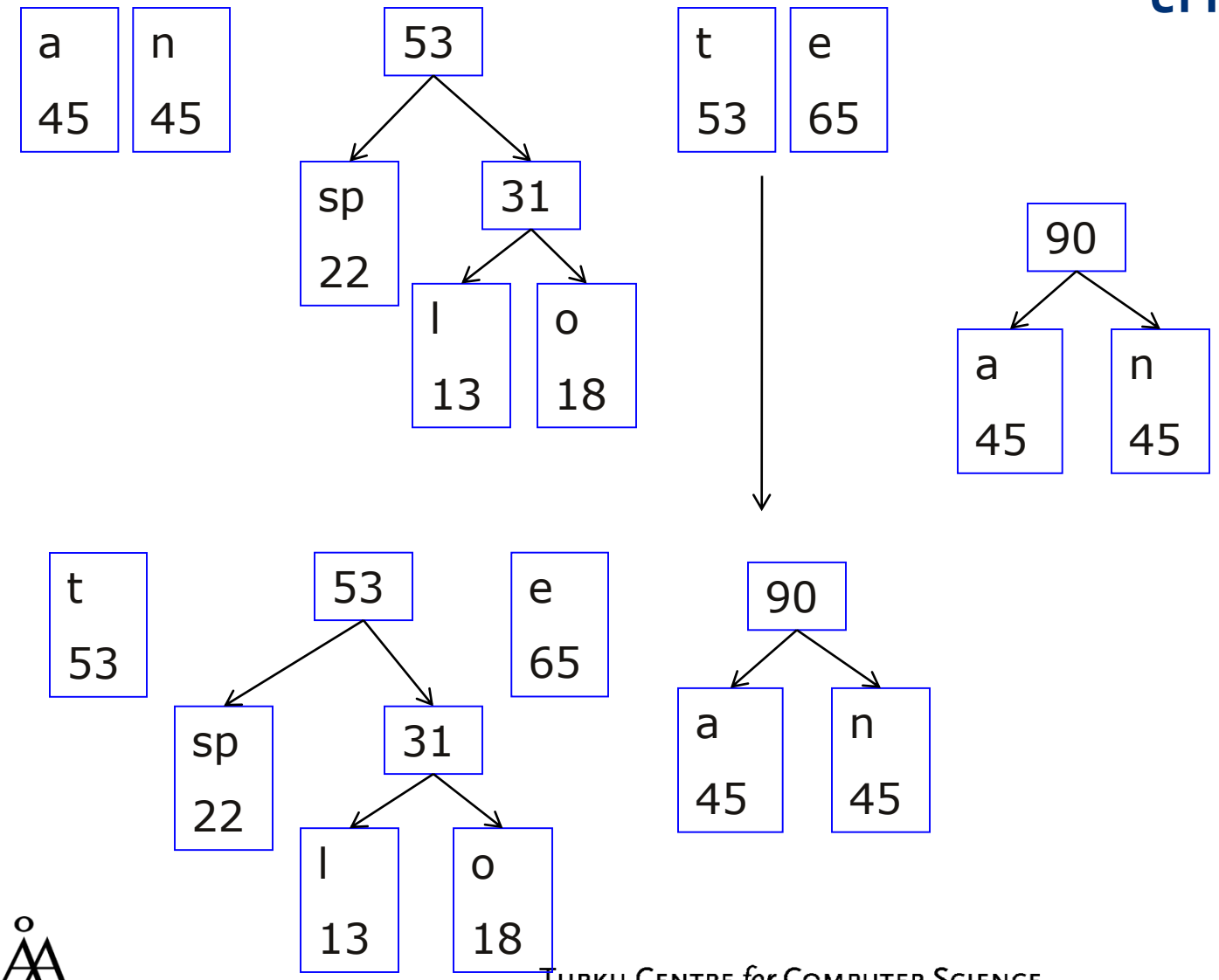
Example for the construction of the tree (3)



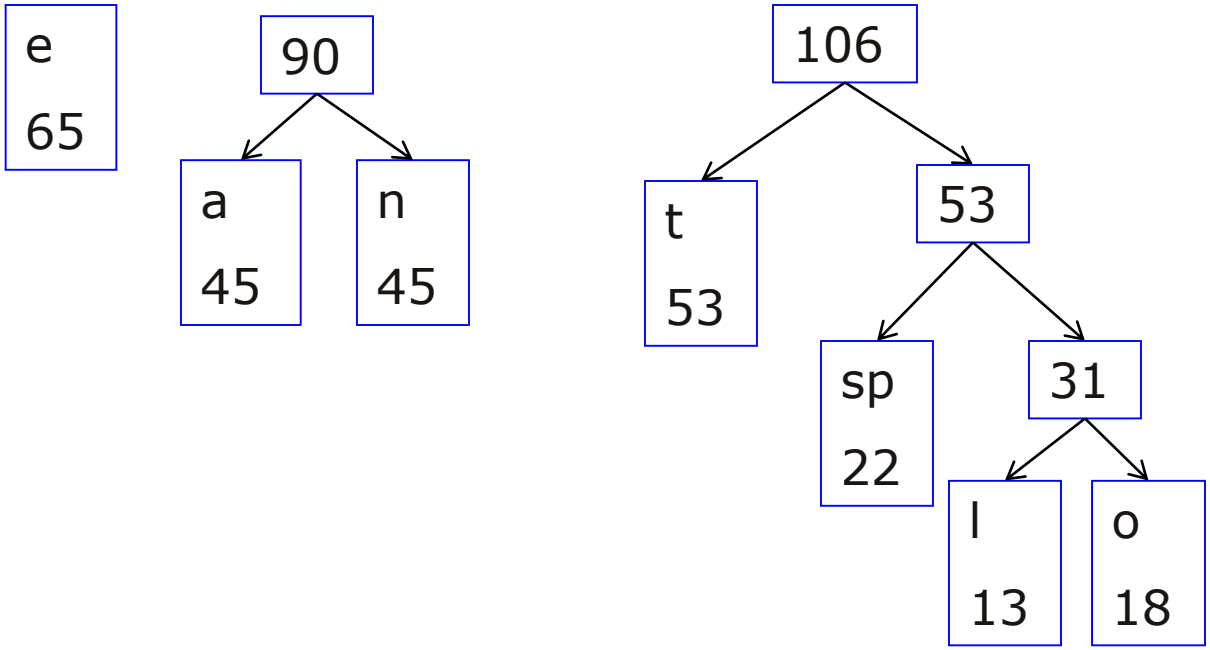
Example for the construction of the tree (4)



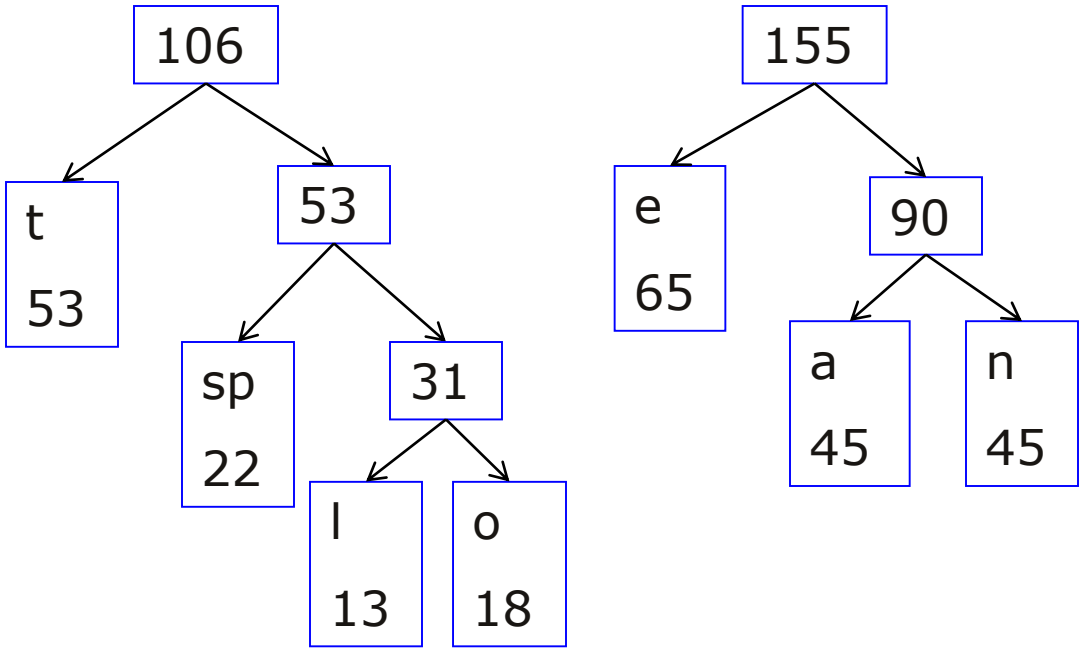
Example for the construction of the tree (5)



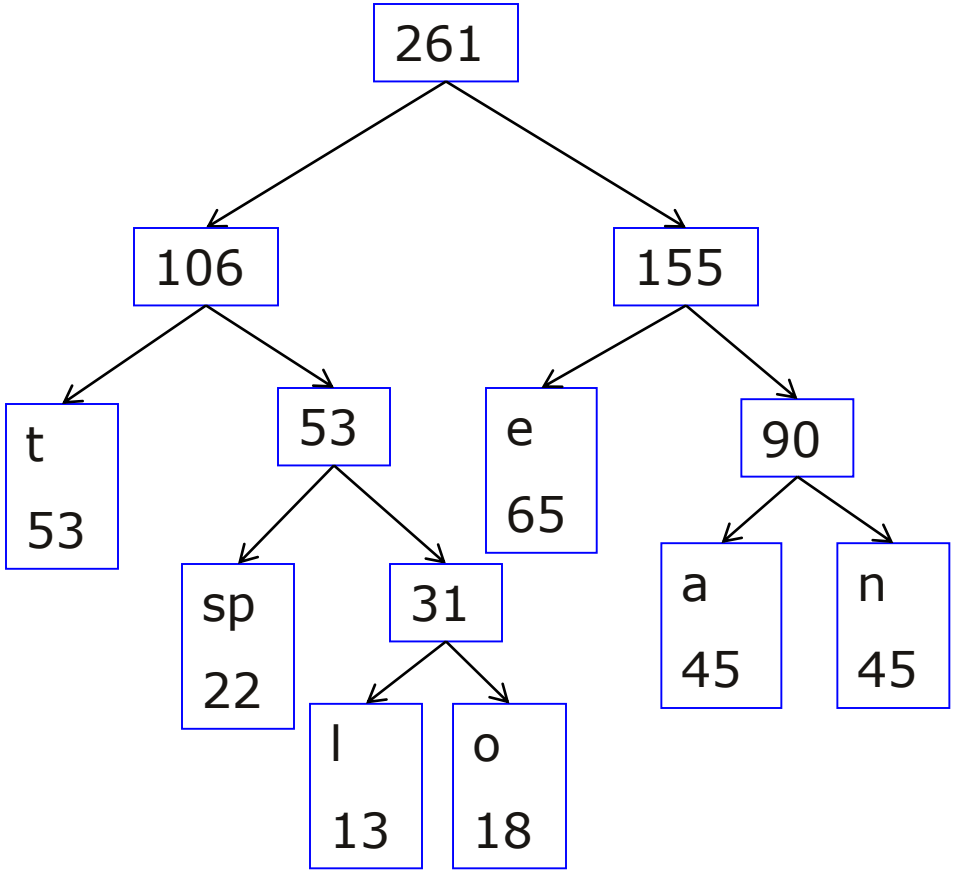
Example for the construction of the tree (6)



Example for the construction of the tree (7)



Example for the construction of the tree (8)



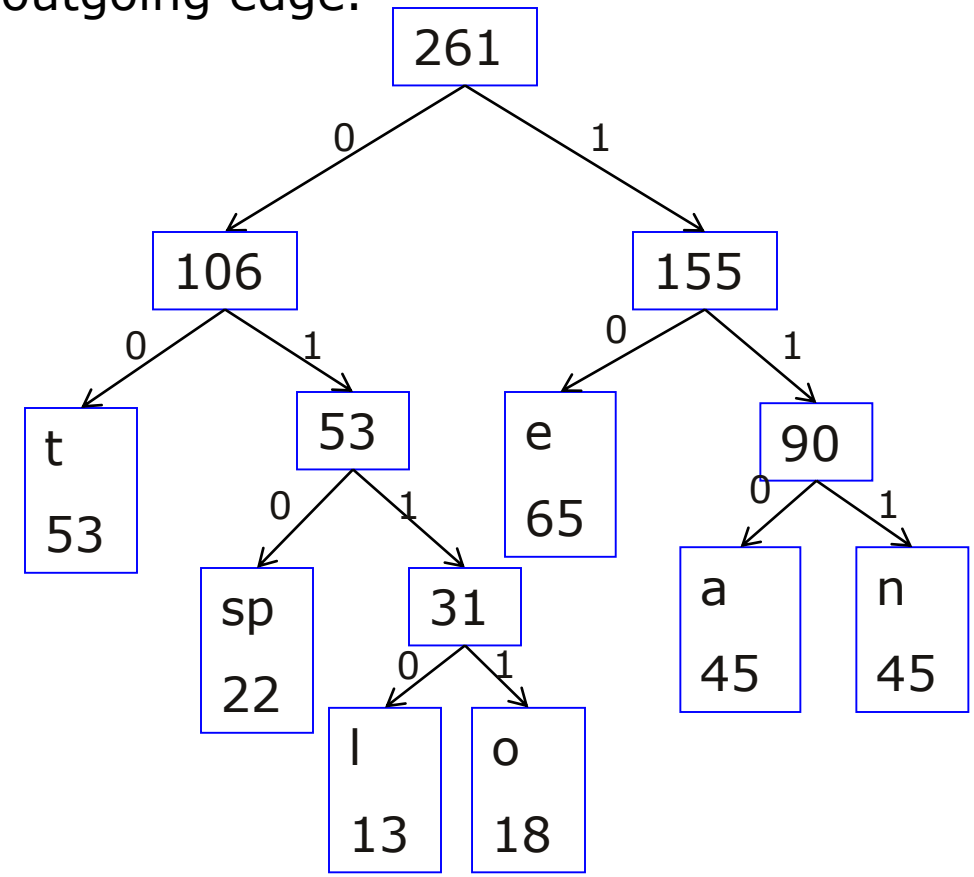
The construction of the tree

- The Huffman code tree for a particular set of characters is not unique.
  - At some moment, in the priority queue, we might have more than 2 trees with the same priority.
  - Depending on which 2 trees we select at that step we will obtain a different final code tree

- However, any of the obtained Huffman code trees are optimal.

Huffman coding

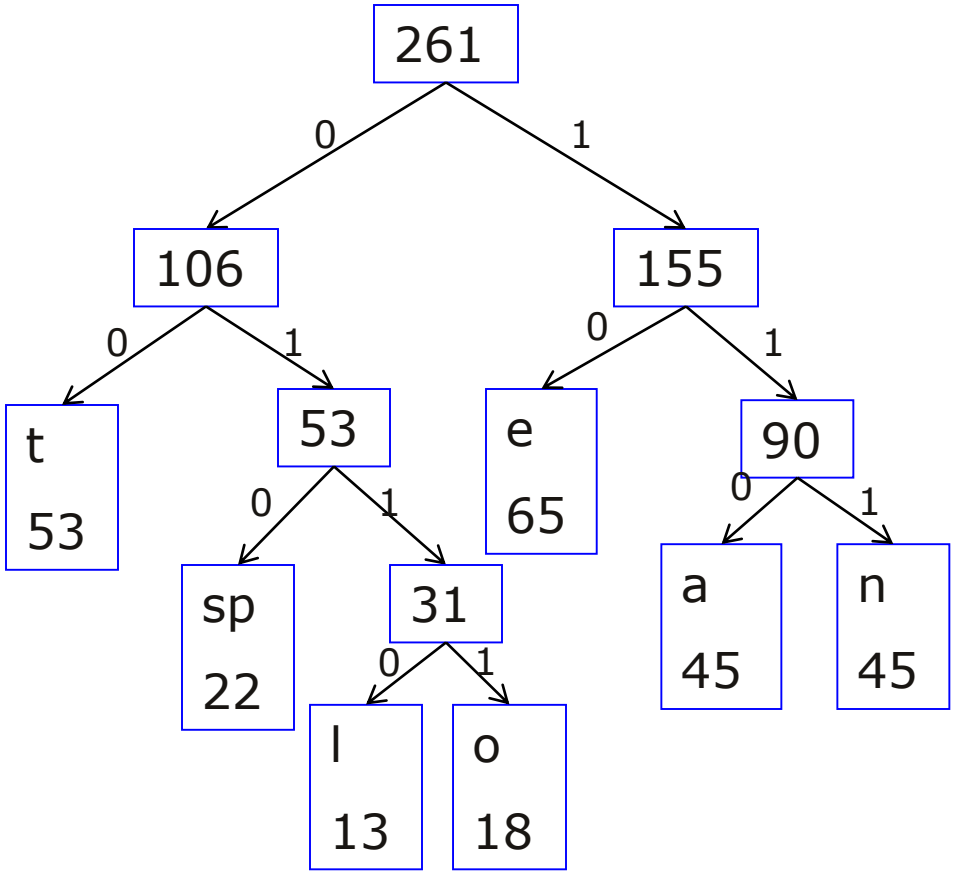
- When we have only one tree in the list of priorities, we assign 0 or 1 to each edge:
  - For instance, we assign 0 to each left outgoing edge and 1 to each right outgoing edge.



Huffman coding

- The sequences of zeros and ones that constitute the paths from the root to each leaf node are the desired codewords:

Char.	Codeword
t	00
space	010
l	0110
o	0111
e	10
a	110
n	111



Huffman coding

- Finally, we can scan the text again and replace each letter with its associated codeword to obtain the encoding.

- If we assume that the message contains only these 7 characters a, e, l, n, o, t, and space, with the given frequencies, then the encoded message will have 696 bits:
  - 45 times letter a which is encoded by 110: 45×3=135
  - 65 times letter e which is encoded by 10: 65×2=130
  - 13 times letter l which is encoded by 0110: 13×4=52
  - 45 times letter n which is encoded by 111: 45×3=135
  - 18 times letter o which is encoded by 0111: 18×4=72
  - 22 times letter space which is encoded by 010: 22×3=66
  - 53 times letter t which is encoded by 00: 53×2=106

- If the message is sent uncompressed with 8-bit ASCII representation for the characters 8 * 261= 2088 bits

Huffman coding

Assuming that the number of character-codeword pairs and the pairs are included at the beginning of the binary file containing the compressed message in the following format:

7 ← in binary (significant bits)
a110 ← characters are in 8-bit ASCII codes
e10
l0110
n111
o0111
s010
t00
sequence of zeroes and ones for the compressed message

Number of bits for the encoded transmitted file = bits(7) + bits(characters) + bits(codewords) + bits(compressed message) = 3 + (7*8) + 21 + 696 = 776
Thus the size of the encoded message is 37% of the original ASCII file.

Encoding using Huffman algorithm

- Encode the word talent using the following codewords:

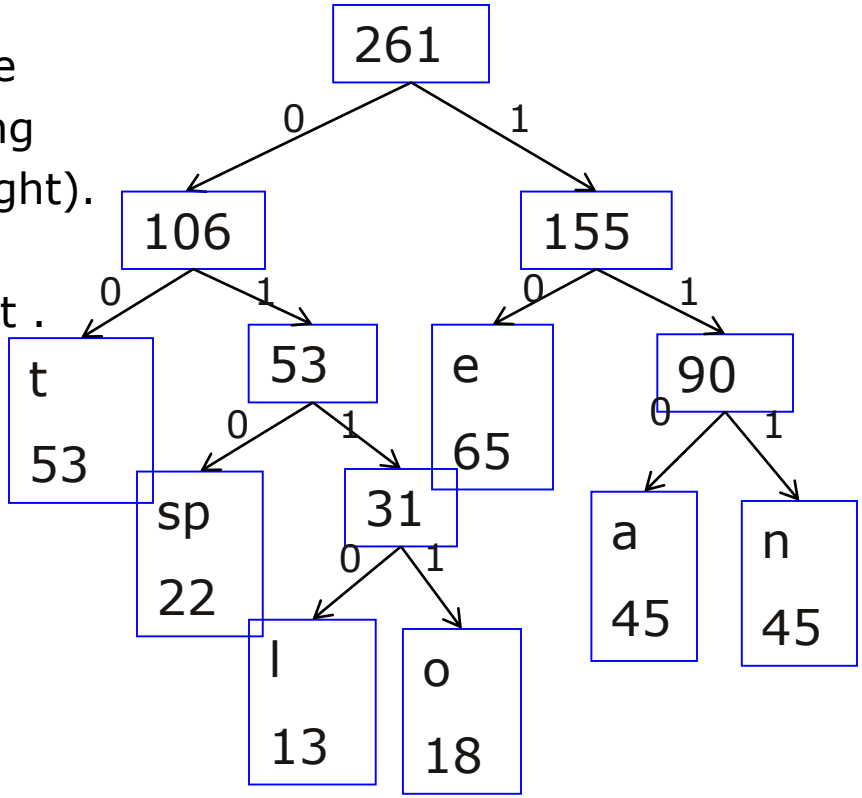
Char.	Codeword
t	00
space	010
l	0110
o	0111
e	10
a	110
n	111

- Replace each letter with the associated codeword: 00110011010111100

Decoding using Huffman algorithm

- Decode the following encoded messages, if possible, using the Huffman codeword tree given below **0110011101000** and **11101110101011**:

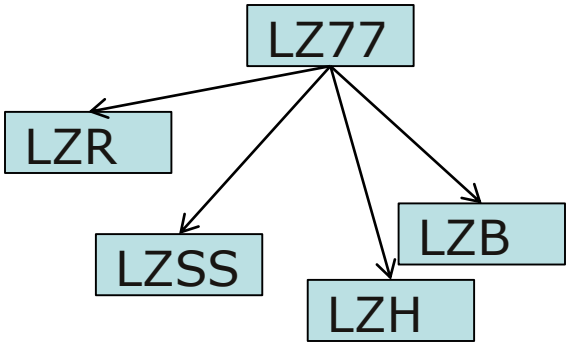
- Decode a bit-stream by starting at the root and proceeding down the tree according to the bits in the message (0 = left, 1 = right). When a leaf is encountered, output the character at that leaf and restart at the root . If a leaf cannot be reached, the bit-stream cannot be decoded.
- **0110011101000** → **lost**
- **11101110101011** → **no_e** but the decoding fails because the corresponding node for 11 is not a leaf



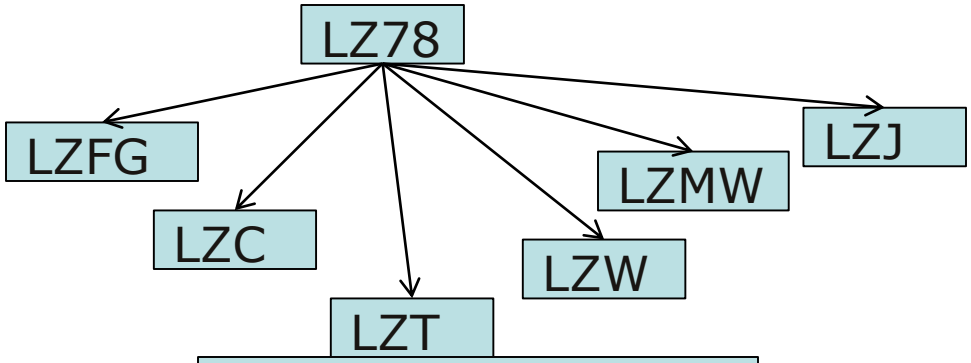
Lempel-Ziv compression algorithms

Lempel-Ziv compression algorithms

- Lempel-Ziv technique is another method for lossless data compression.
- It relies on the fact that, in any document, character strings are going to be repeated.
- It actually consists of a whole family of algorithms, stemming from two algorithms proposed by Jacob Ziv and Abraham Lempel in 1977 and 1978.



Applications:
- zip
- gzip
- ...



Applications:
- GIF
- UNIX's **compress** method
- ...

Lempel-Ziv 78 compression algorithm

- The LZ78 is a dictionary-based compression algorithm that maintains an explicit dictionary.
- The codewords output by the algorithm consist of two elements: an index referring to the longest matching dictionary entry and the first non-matching symbol.
- This algorithm is simple to implement and has become popular as one of the early standard algorithms for file compression on computers because of its speed and efficiency.
- It is also used for data compression in high-speed modems.

Lempel-Ziv 78 compression algorithm

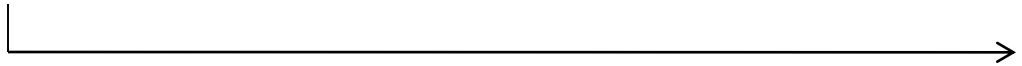
```
Dictionary ← empty ; Prefix ← empty ; DictionaryIndex  ← 1;
while(characterStream is not empty) do begin
    Char ← next character in characterStream;
     if(Prefix + Char exists in the Dictionary)
           Prefix ← Prefix + Char ;
      else do begin
            if(Prefix is empty)
                 CodeWordForPrefix  ← 0 ;
           else
                 CodeWordForPrefix ← DictionaryIndex for Prefix ;
           Output: (CodeWordForPrefix, Char) ;
           insertInDictionary( ( DictionaryIndex , Prefix + Char) );
           DictionaryIndex++ ;
           Prefix ← empty ;
      end
end
if(Prefix is not empty) do begin
    CodeWordForPrefix ← DictionaryIndex for Prefix;
    Output: (CodeWordForPrefix ,   ) ;
end
```

Example for LZ78 coding

Take the text ABBCBCABABCAABCAAB

- We start with an empty dictionary
- The first character is A, which is not in the dictionary, so we introduce it:

ABBCBCABABCAABCAAB



Dictionary

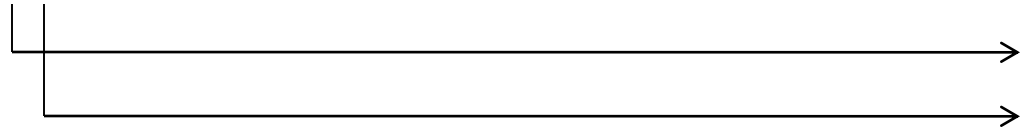
index	string	Output
1	A	(0,A)

The output is the pair (0,A):
-0 is the index of the maximal prefix of the string A which is already in the dictionary: ε
-A is the next character of the string A following the prefix found in the dictionary

Example for LZ78 coding(2)

- Then we read the next character B, which is not in the dictionary, so we introduce it:

ABBCBCABABCAABCAAB



Dictionary

index	string	Output
1	A	(0,A)
2	B	(0,B)

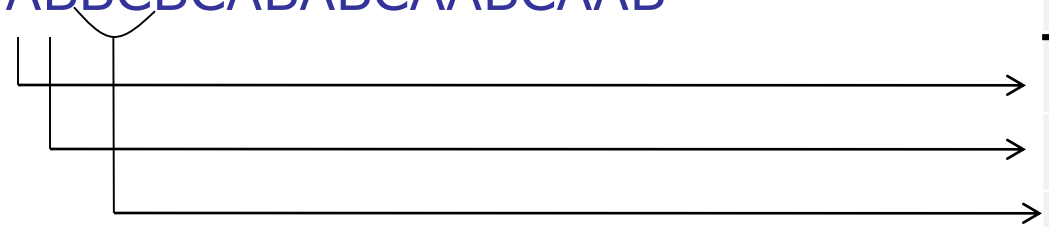
The output is the pair (0,B):

-0 is the index of the maximal prefix of the string B which is already in the dictionary: ε

-B is the next character of the string B following the prefix found in the dictionary

Example for LZ78 coding(3)

- Then we read the next character B, which is already in the dictionary
- So we read the next character C, and the obtained string BC is not in the dictionary so we introduce it



Dictionary

index	string	Output
1	A	(0,A)
2	B	(0,B)
3	BC	(2,C)

The output is the pair (2,C):

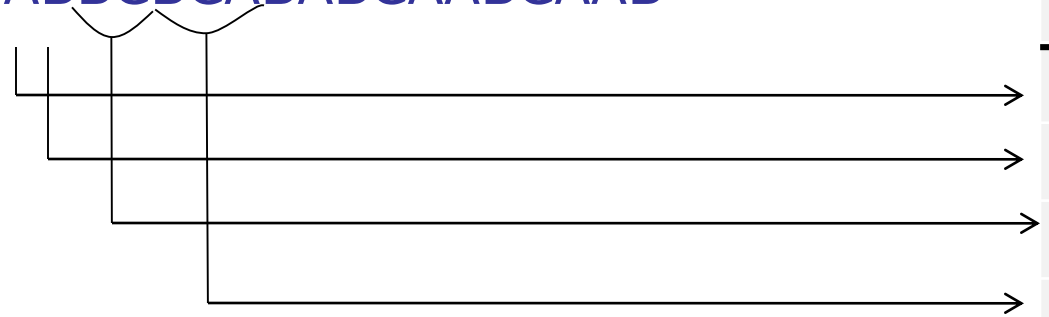
-2 is the index of the maximal prefix of the string BC which is already in the dictionary: B

-C is the next character of the string BC following the prefix found in the dictionary

Example for LZ78 coding(4)

- Then we read the next character B, which is already in the dictionary
- So we read the next character C, and the obtained string BC is in the dictionary
- So we read the next character A, and we introduce BCA in the dictionary

ABBCBCABABCAABCAAB

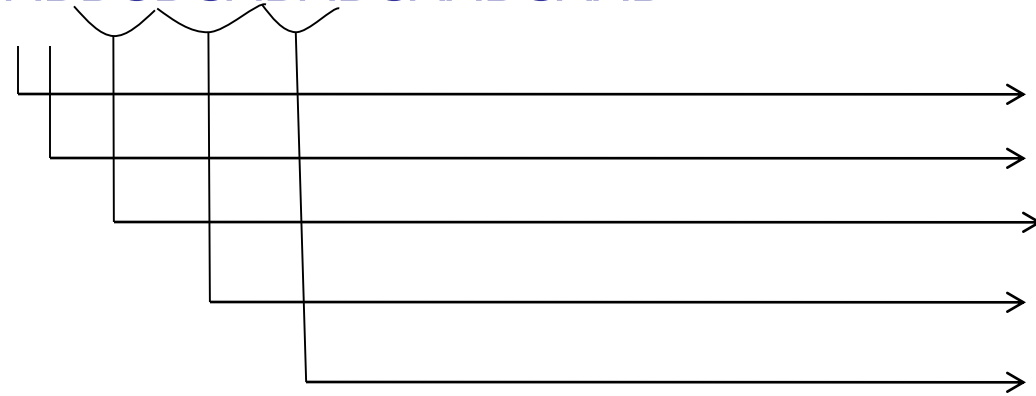


Dictionary

index	string	Output
1	A	(0,A)
2	B	(0,B)
3	BC	(2,C)
4	BCA	(3,A)

Example for LZ78 coding(5)

- Then we read the next character B, which is already in the dictionary
- So we read the next character A, and we introduce in the dictionary the obtained string BA

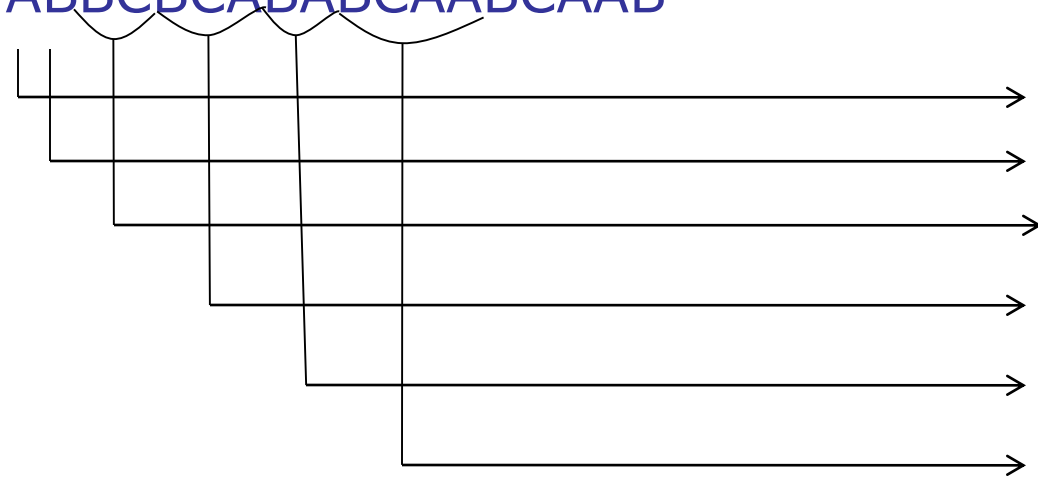


Dictionary

index	string	Output
1	A	(0,A)
2	B	(0,B)
3	BC	(2,C)
4	BCA	(3,A)
5	BA	(2,A)

Example for LZ78 coding(6)

- Then we read character by character the string BCA: all strings B, BC, and BCA are already in the dictionary
- So, we read the next character A, and we introduce BCAA in the dictionary

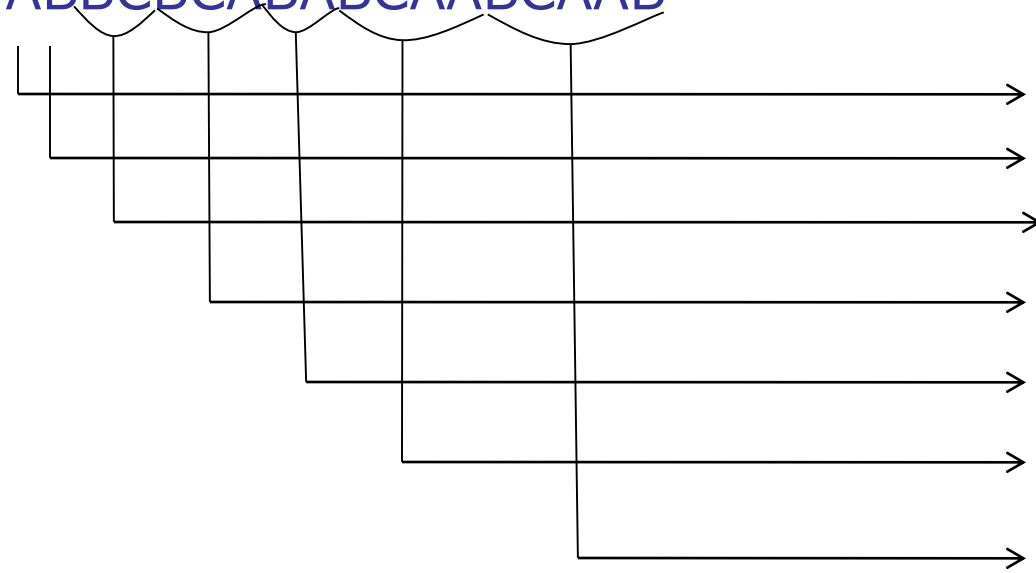


Dictionary

index	string	Output
1	A	(0,A)
2	B	(0,B)
3	BC	(2,C)
4	BCA	(3,A)
5	BA	(2,A)
6	BCAA	(4,A)

Example for LZ78 coding(7)

- Then we read character by character the string BCAA: all strings B, BC, BCA, BCAA are already in the dictionary
- So, we read the next character B, and we introduce BCAAB in the dictionary



Dictionary

index	string	Output
1	A	(0,A)
2	B	(0,B)
3	BC	(2,C)
4	BCA	(3,A)
5	BA	(2,A)
6	BCAA	(4,A)
7	BCAAB	(6,B)

Example for LZ78 coding (8)

- Thus, the compressed message associated to the text ABBCBCABABCAABCAAB is (0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)

- Actually, this is just a representation of the coded message:
  - The commas and the parenthesis are not included in the compressed message
  - Each code pair (k,A) with index i is actually transmitted as follows:
    - The letter is represented by its 8-bits ASCII code
    - The base 2 representation of k is used instead of k, and moreover, the number n of bits used in this representation is:

$$n = \begin{cases} 1, & \text{if } i = 1 \\ \lceil \log_2 i \rceil, & \text{if } i > 1 \end{cases}$$

Example for LZ78 coding(9)

- For the text ABBCBCABABCAABCAAB we have

Codewords: (0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)
with

Indexes: 1 2 3 4 5 6 7

Thus, the compressed message is:

0ASCII(A)0ASCII(B)10ASCII(C)11ASCII(A)010ASCII(A)100
ASCII(A)110ASCII(B)

- Observation: The number of bits required to represent the integer part of the codeword with index **i** is the number of significant bits required to represent the integer **i – 1**

Compression efficiency

- The number of bits used in the uncompressed string ABBCBCABABCAABCAAB is 18×8=144

- The number of bits used in the compressed message

0ASCII(A)0ASCII(B)10ASCII(C)11ASCII(A)010ASCII(A)
100ASCII(A)110ASCII(B)

is 15+7×8=71 which represents 49% of the original text

LZ78 decoding algorithm

```
Dictionary ← empty ; DictionaryIndex ← 1 ;
while(there are more (CodeWord, Char) pairs in codestream) {
    CodeWord ← next CodeWord in codestream ;
    Char ← character corresponding to CodeWord ;
    if(CodeWord == 0)
            String ← empty ;
    else
        String ← string at index CodeWord in Dictionary ;
    Output: String + Char ;
    insertInDictionary( (DictionaryIndex ,  String + Char) ) ;
      DictionaryIndex++;
}
```

Example for LZ78 coding

Take the sequence (0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)

- We start with the empty dictionary

We take the first code pair (0,A): CodeWord ← 0; Char ← A

(0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)

We output the current character A and we introduce in the dictionary the concatenation of the string at index 0 (i.e., ε) and of the current character

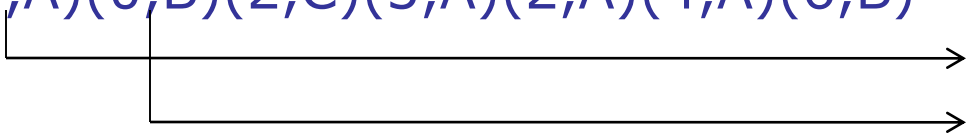
Dictionary

index	string	Output
1	A	A

Example for LZ78 coding

We read the next code pair (0,B): CodeWord ← 0; Char ← B

(0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)



Dictionary

index	string
1	A
2	B

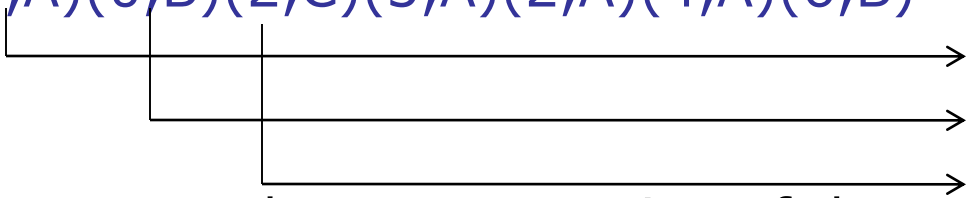
Output
A
B

We output the current character B and we introduce in the dictionary the concatenation of the string at index 0 (i.e., ε) and of the current character

Example for LZ78 coding

We read the next code pair (2,C): CodeWord ← 2; Char ← C

(0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)



We output the concatenation of the string at index 2 (i.e., B) and of the current character (i.e., C) and
We also introduce BC in the dictionary.

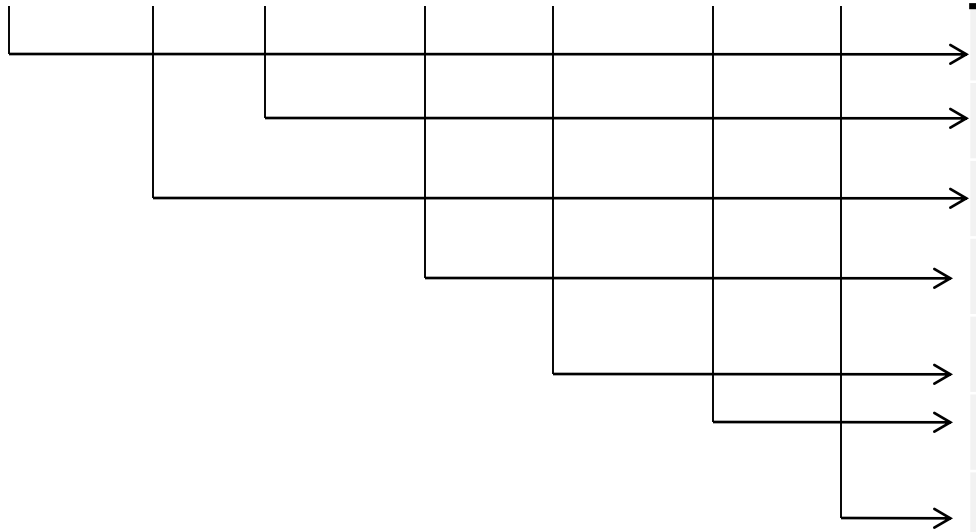
Dictionary

index	string
1	A
2	B
3	BC

Output
A
B
BC

Example for LZ78 coding

(0,A)(0,B)(2,C)(3,A)(2,A)(4,A)(6,B)



Dictionary

index	string
1	A
2	B
3	BC
4	BCA
5	BA
6	BCAA
7	BCAAB

Output
A
B
BC
BCA
BA
BCAA
BCAAB

Thus, the decoded message is ABBCBCABABCAABCAAB