University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 6: this Friday @ 11:59 pm
  - Lab 5: next Monday @ 11:59 pm
  - Assignment 1 Late Deadline Wednesday Oct 12th @ 11:59 pm
- Midterm Exam
  - Wednesday 10/19 (MW Section) and Thursday 10/20 (TuTh Section)
    - in-person, closed-book
- If you think you lost points in a lab assignment because of the autograder or because of a simple mistake
  - please reach out to Grader TA over Piazza
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous lecture

- Huffman Compression
  - How to compute character frequencies
- Run-length Encoding
- LZW
  - compression and expansion algorithms

This Lecture

- LZW
  - implementation concerns
- Shannon's Entropy
- Comparing LZW vs Huffman
- Burrows-Wheeler Compression Algorithm

LZW implementation concerns: codebook

- How to represent/store during:
  - Compression
  - Expansion
- Considerations:
  - What operations are needed?
  - How many of these operations are going to be performed?
- Discuss

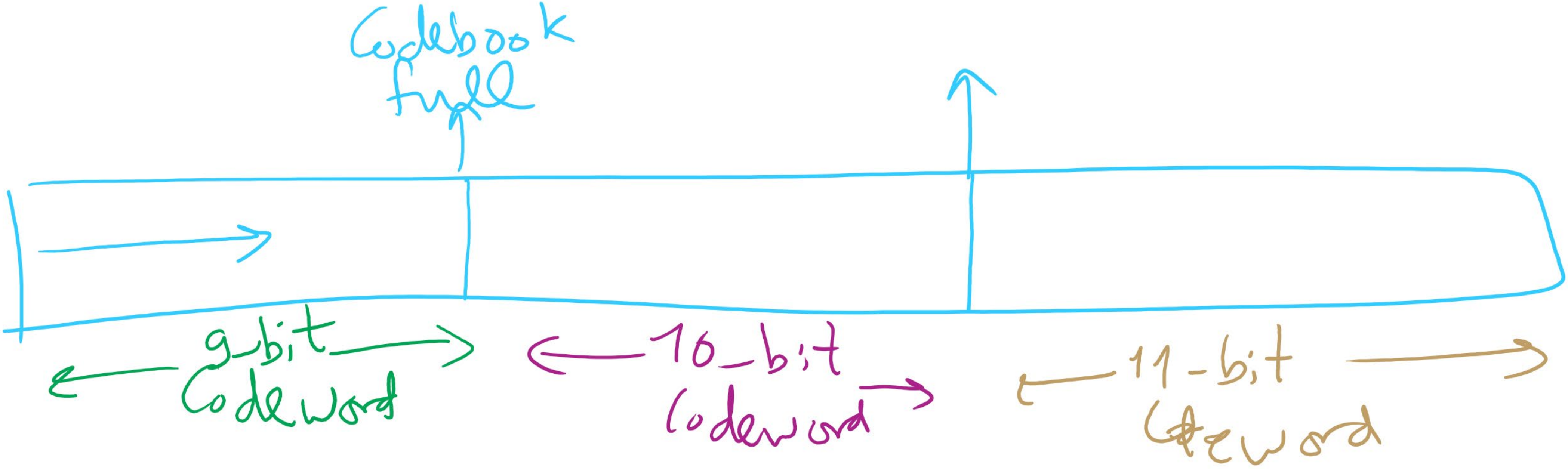
Further implementation issues: codeword size

- How long should codewords be?
  - Use fewer bits:
    - Gives better compression earlier on
    - But, leaves fewer codewords available, which will hamper compression later on
  - Use more bits:
    - Delays actual compression until longer patterns are found due to large codeword size
    - More codewords available means that greater compression gains can be made later on in the process

Variable width codewords

- This sounds eerily like variable length codewords…
  - Exactly what we set out to avoid!
- Here, we're talking about a different technique
- Example:
  - Start out using 9 bit codewords
  - When codeword 512 is inserted into the codebook, switch to outputting/grabbing 10 bit codewords
  - When codeword 1024 is inserted into the codebook, switch to outputting/grabbing 11 bit codewords…
  - Etc.

Adaptive Codeword Size



Even further implementation issues: codebook size

- What happens when we run out of codewords?
  - Only 2^n possible codewords for n bit codes
  - Even using variable width codewords, they can't grow arbitrarily large…
- Two primary options:
  - Stop adding new keywords, use the codebook as it stands
    - Maintains long already established patterns
    - But if the file changes, it will not be compressed as effectively
  - Throw out the codebook and start over from single characters
    - Allows new patterns to be compressed
    - Until new patterns are built up, though, compression will be minimal

Can we reason about how much a file can be compressed?

- Yes! Using Shannon Entropy

Information theory in a single slide...

- Founded by Claude Shannon in his paper "A Mathematical Theory of Communication"
- *Entropy* is a key measure in information theory
  - Slightly different from thermodynamic entropy
  - A measure of the unpredictability of information content
  - Example: which is more unpredicatble?
    - a character that occurs with a probabillity of 0.5 or
    - a character that occurs with probability 0.25
    - which should have more entropy?

Entropy

- Entropy equation: $H(c) = -1 * \log_2 Pr(c)$
  - ○ $Pr(c)$ is the probability of character c
- Examples:
  - ○ $Pr(c1) = 0.5 \rightarrow H(c1) = -1 * \log_2(0.5) = -1*-1 = 1$ bit
  - ○ $Pr(c2) = 0.25 \rightarrow H(c2) = -1*\log_2(0.25) = -1*-2 = 2$ bits
  - ○ $Pr(c3) = 1/2^{100} \rightarrow H(c3) = -1*\log_2(2^{-100}) = -1*-100 = 100$ bits

Implications on Lossless Compression

- On average, a lossless compression scheme cannot compress a message to have more than 1 bit of entropy per bit of compressed message
- By losslessly compressing data, we represent the same information in less space
  - entropy of 8 bits of compressed data >
    entropy of 8 bits of uncompressed data

Entropy of a file

- The average number of bits required to store a character in that file
- So, it is the average entropy of all unique characters in the file
- H(file) = $\text{sum}_{\text{each unique character } c}$ *H(c)*Pr(c)*
- How can we determine the probability of each character in the file?
  - if depends only on file contents
    - *Pr(c)* = *f(c)* / file size
  - However, may also depend on receiver and sender contexts and their world knowledge

Entropy applied to language:

- the average number of bits required to store a letter of the language
- Entropy of a language * length of message = amount of information contained in that message
- Uncompressed, English has between 0.6 and 1.3 bits of entropy per letter

The showdown you’ve all been waiting for...

HUFFMAN vs LZW

- In general, LZW will give better compression
    - Also better for compressing archived directories of files
        - Why?
            - Very long patterns can be built up, leading to better compression
            - Different files don’t “hurt” each other as they did in Huffman
                - Remember our thoughts on using static tries?

So lossless compression apps use LZW?

- Well, gifs can use it
  - And pdfs
- Most dedicated compression applications use other algorithms:
  - DEFLATE (combination of LZ77 and Huffman)
    - Used by PKZIP and gzip
  - Burrows-Wheeler transforms
    - Used by bzip2
  - LZMA
    - Used by 7-zip
  - brotli
    - Introduced by Google in Sept. 2015
    - Based around a " … combination of a modern variant of the LZ77 algorithm, Huffman coding[,] and 2nd order context modeling … "

Is there a univeral compression algorithm?

- Nope!
- No algorithm can compress every bitstream
  - Assume we have such an algorithm
  - We can use to compress its own output!
  - And we could keep compressing its output until our compressed file is 0 bits!
    - Clearly this can't work
- Proofs in Proposition S of Section 5.5 of the text

Is finding the best algorithm for a given file possible?

- Nope!
- This problem is undecidable
- Example:
    - A Fibonacci sequence of one billion numbers can be compressed by a program to generate Fibonacci numbers

A final note on compression evaluation

- "Weissman scores" are a made-up metric for Silicon Valley (TV)

Burrows-Wheeler Data Compression Algorithm

- Best compression algorithm (in terms of compression ratio) for text
- The basis for UNIX’s bzip2 tool

Adapted from: https://www.cs.princeton.edu/courses/archive/spr03/cos226/assignments/burrows.html

BWT: Compression Algorithm

- Three steps
  - Cluster same letters as close to each other as possible
    - Burrows-Wheeler Transform
  - Move-To-Front Encoding
    - Convert output of previous step into an integer file with large frequency differences
  - Huffman Compression
    - Compress the file of integers using Huffman

BWT: Expansion Algorithm

- Apply the inverse of compression steps in reverse order
  - Huffman decoding
  - Move-To-Front decoding
  - Inverse Burrows-Wheeler Transform

Move-To-Front Encoding

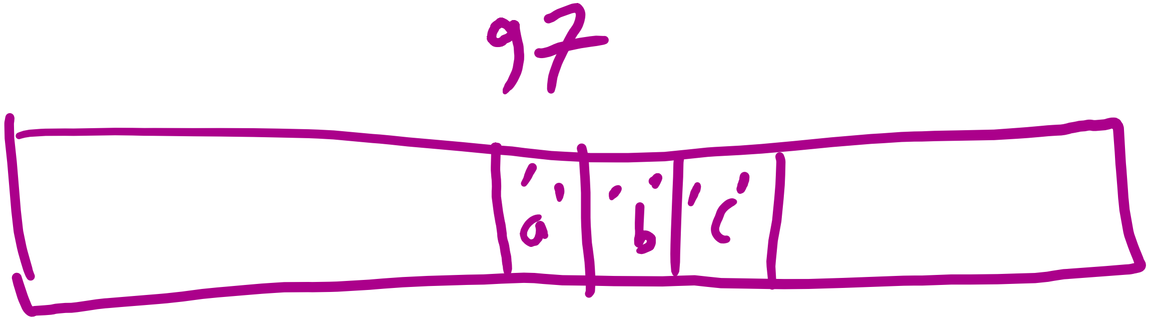
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97

- 'a' is 97 in ASCII



Move-To-Front Encoding

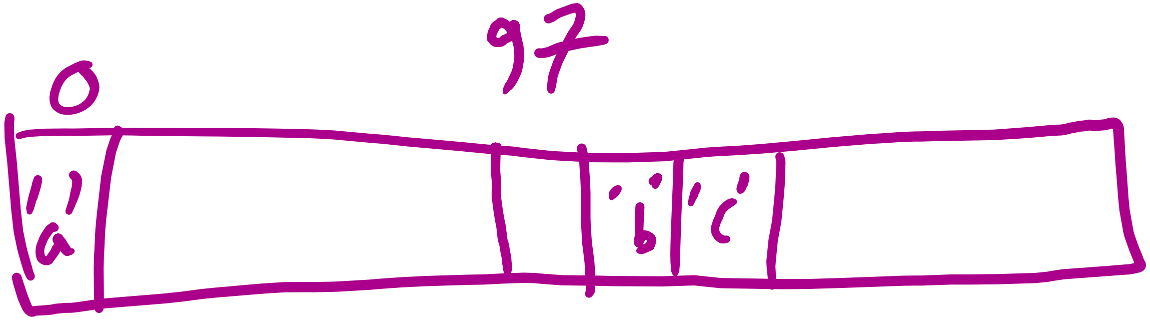
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97

- 'a' is 97 in ASCII



Move-To-Front Encoding

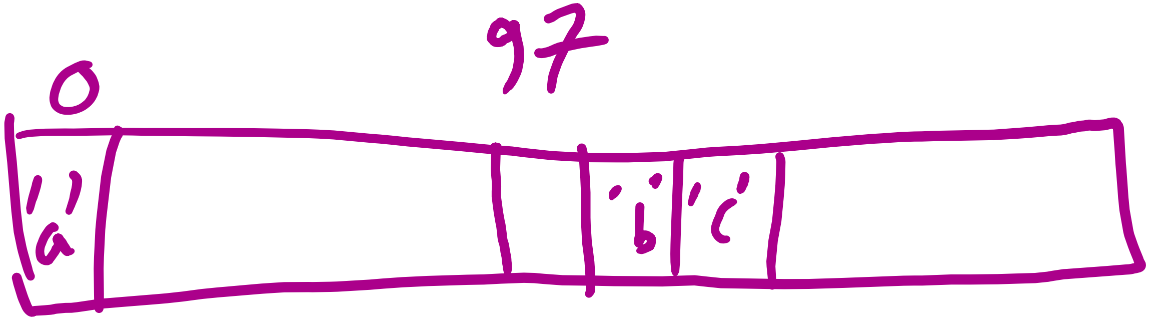
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a **b** b b a a b b b b a c c a b b a a a b c

97 98

- 'a' is 97 in ASCII



Move-To-Front Encoding

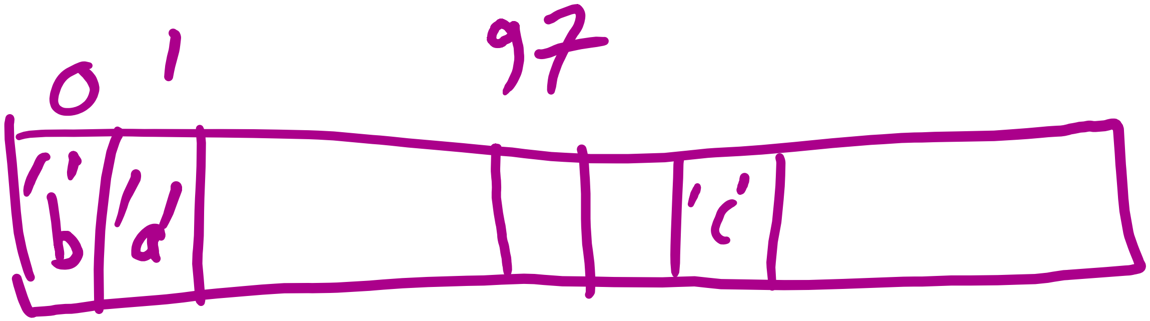
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a **b** b b a a b b b b a c c a b b a a a b c

97 98

- 'a' is 97 in ASCII



Move-To-Front Encoding

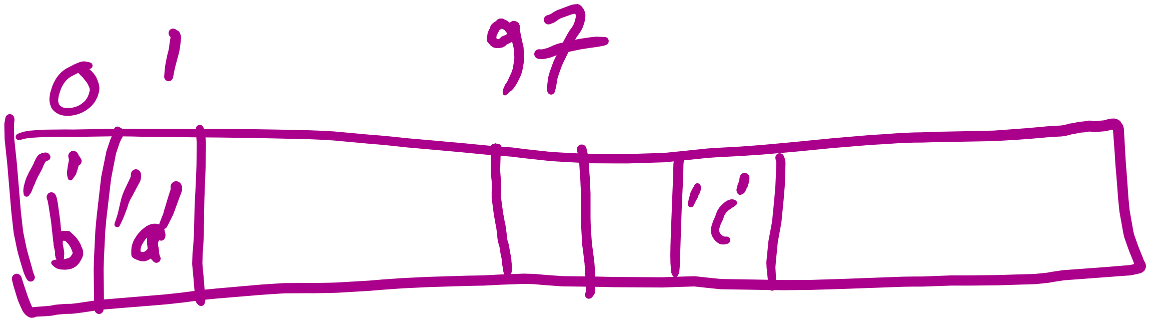
- Initialize an ordered list of the 256 ASCII characters
    - extended ASCII character *i* appears *i*th in the list
- For each character c from input
    - output the index in the list where c appears
    - move c to the front of the list

- Example:

a b **b** b a a b b b b a c c a b b a a a b c

97 98 0

- 'a' is 97 in ASCII



Move-To-Front Encoding

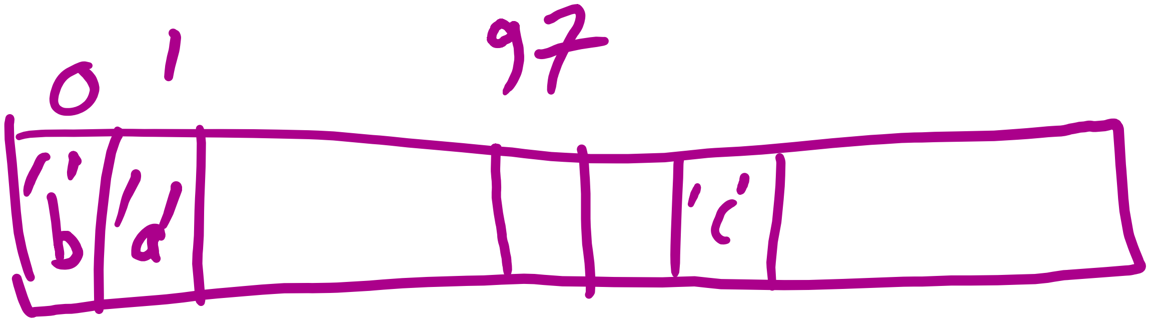
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b **b** a a b b b b a c c a b b a a a b c

97 98 0 0

- 'a' is 97 in ASCII



Move-To-Front Encoding

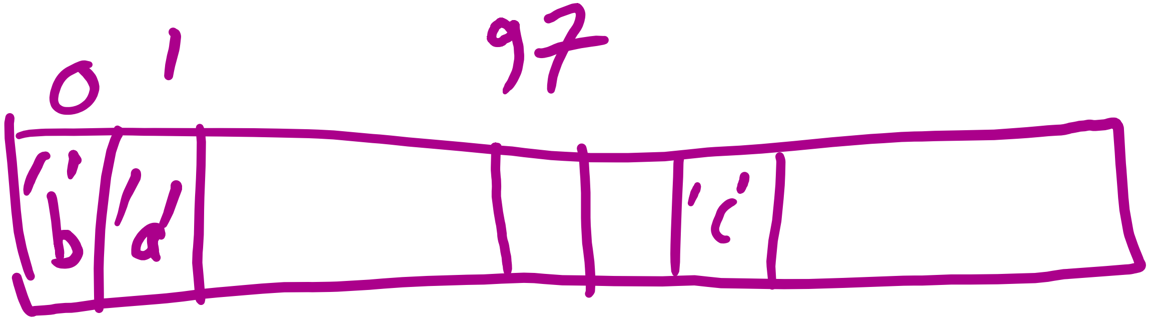
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b **a** a b b b b a c c a b b a a a b c

97 98 0 0 1

- 'a' is 97 in ASCII



CS 1501 – Algorithms & Data Structures 2 – Sherif Khattab

Move-To-Front Encoding

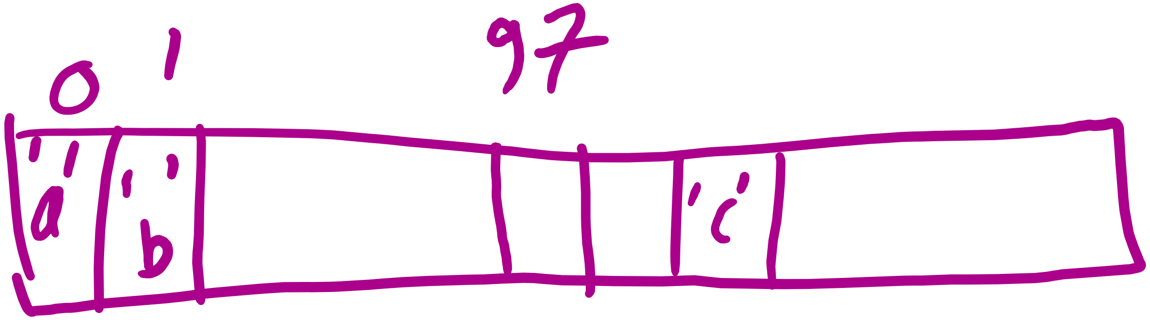
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b **a** a b b b b a c c a b b a a a b c

97 98 0 0 1

- 'a' is 97 in ASCII



Move-To-Front Encoding

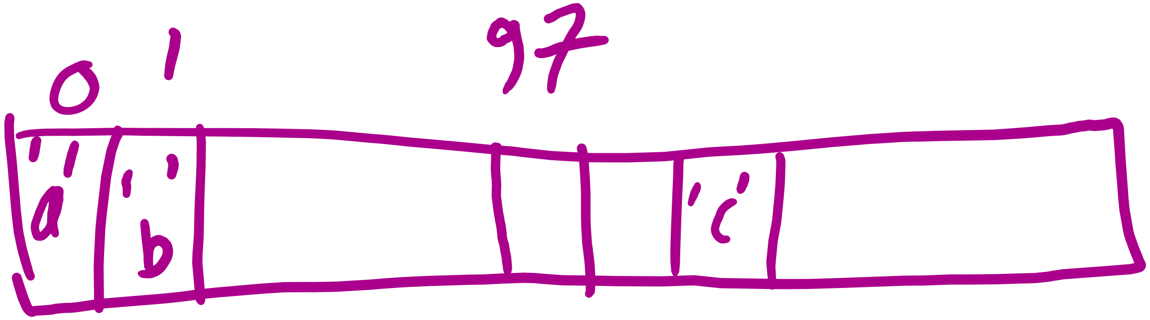
- Initialize an ordered list of the 256 ASCII characters
    - extended ASCII character *i* appears *i*th in the list
- For each character c from input
    - output the index in the list where c appears
    - move c to the front of the list

- Example:

a b b b a **a** b b b b a c c a b b a a a b c

97 98 0 0 1 0

- 'a' is 97 in ASCII



Move-To-Front Encoding

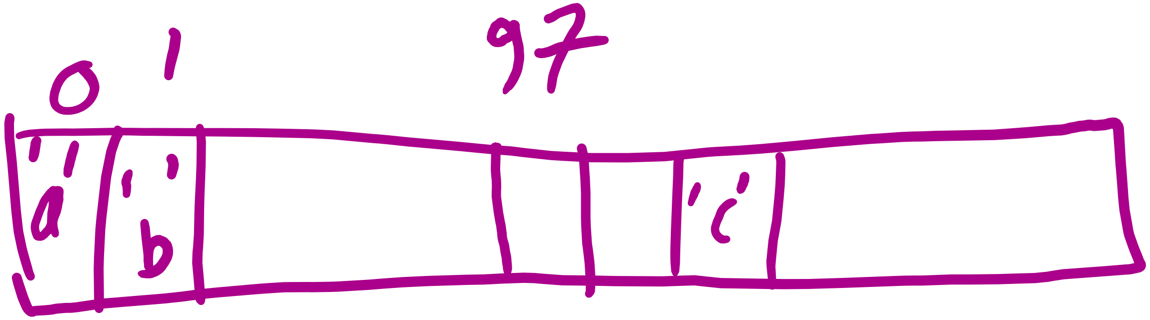
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a **b** b b b a c c a b b a a a b c

97 98 0 0 1 0 1

- 'a' is 97 in ASCII



Move-To-Front Encoding

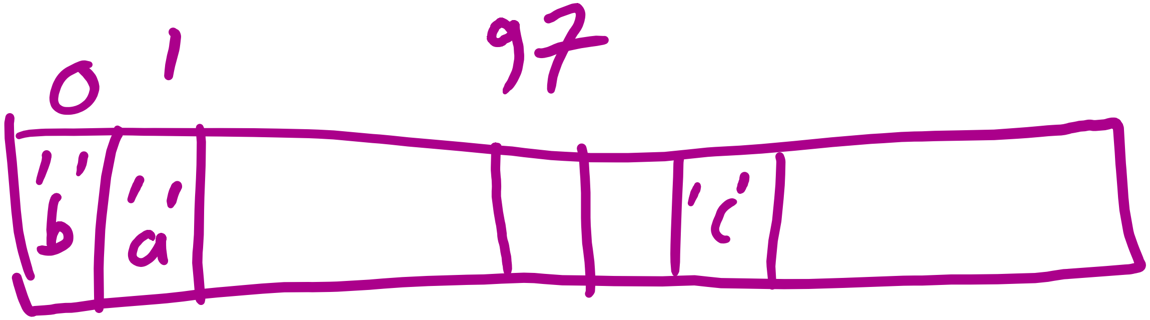
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a **b** b b b a c c a b b a a a b c

97 98 0 0 1 0 1

- 'a' is 97 in ASCII



Move-To-Front Encoding

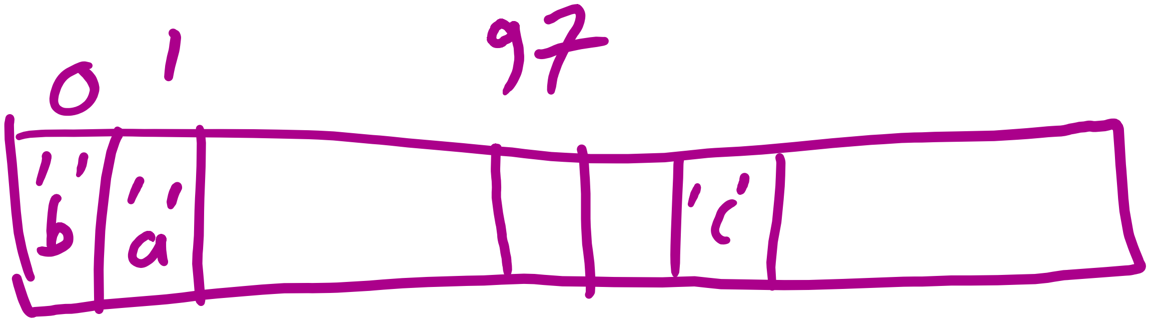
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b **b b b** a c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0

- 'a' is 97 in ASCII



Move-To-Front Encoding

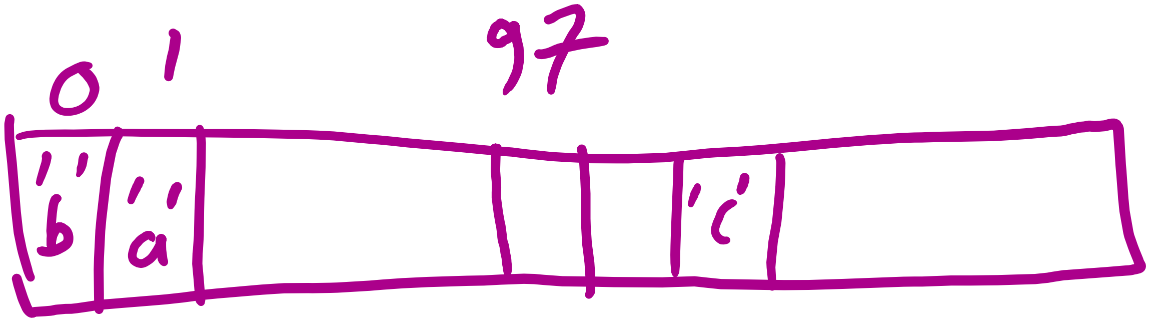
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b b b b **a** c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1

- 'a' is 97 in ASCII



Move-To-Front Encoding

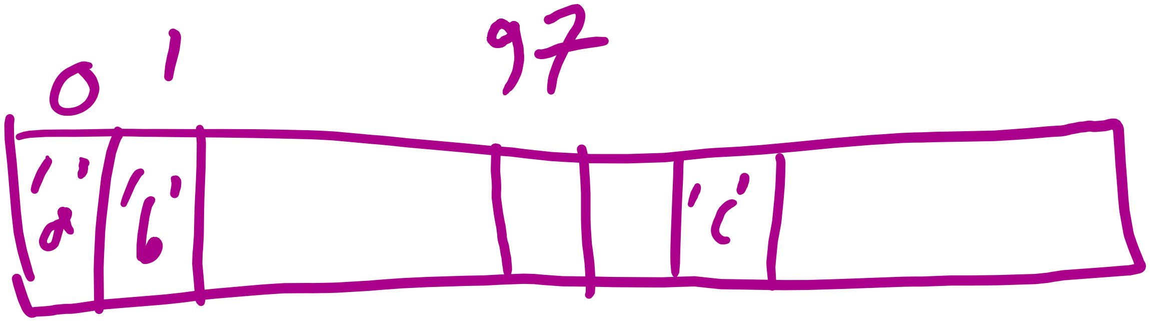
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b b b b **a** c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1

- 'a' is 97 in ASCII



Move-To-Front Encoding

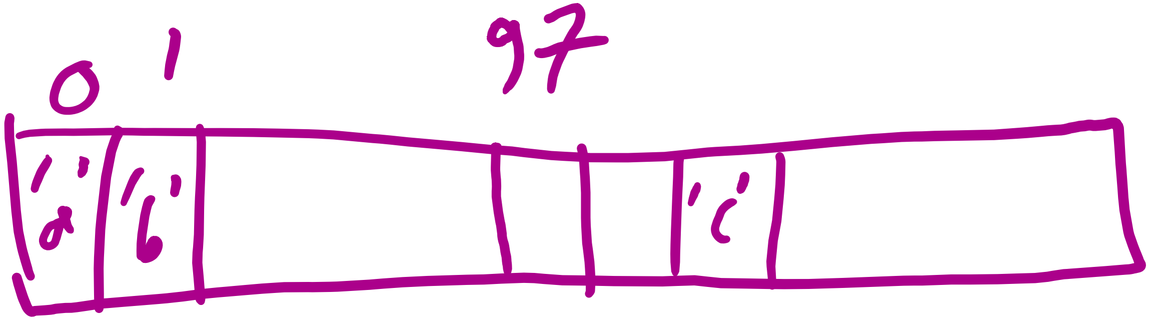
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b b b b a **c** c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1 99

- 'a' is 97 in ASCII



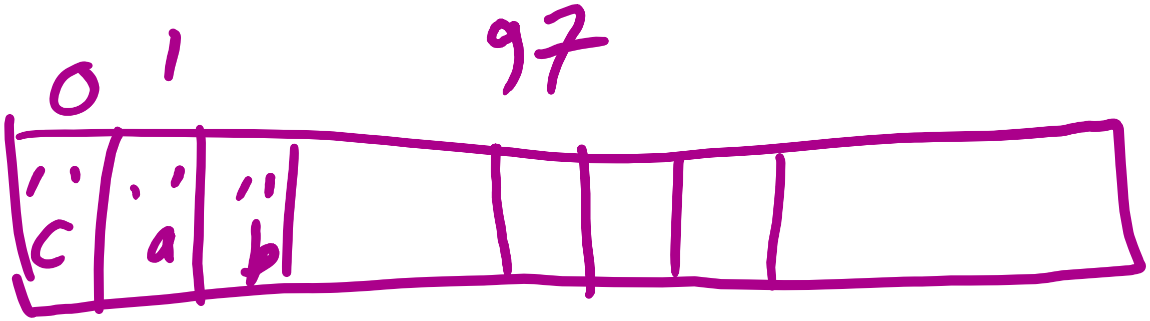
Move-To-Front Encoding

- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list
- Example:

a b b b a a b b b b a **c** c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1 99

- 'a' is 97 in ASCII



Move-To-Front Encoding

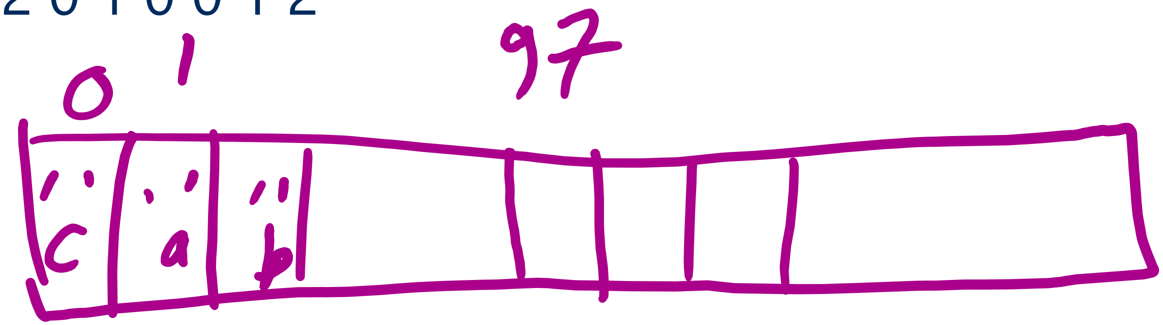
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list

- Example:

a b b b a a b b b b a c **c a b b a a a b c**

97 98 0 0 1 0 1 0 0 0 1 99 0 1 2 0 1 0 0 1 2

- 'a' is 97 in ASCII



Move-To-Front Encoding

In the output of MTF Encoding, smaller integers have higher frequencies than larger integers

Move-To-Front Decoding

- Initialize an ordered list of 256 characters
    - same as encoding
- For each integer *i* (*i* is between 0 and 255)
    - print the *i*th character in the list
    - move that character to the front of the list

Burrows-Wheeler Transform

- Rearranges the characters in the input
    - lots of clusters with repeated characters
    - still possible to recover the original input
- Intuition: Consider **hen** in English text
    - most of the time the letter preceding it is t or w
    - group all such preceding letters together (mostly t's and some w’s)

Burrows-Wheeler Transform

- For each block of length N
  - generate **N strings** by cycling the characters of the block one step at a time
  - sort the strings
  - output is the last column in the sorted table and the index of the original block in the sorted array

Burrows-Wheeler Transform

- Example: Let’s transform “ABRACADABRA”

- N = 11

- Cyclic Versions of the string:
  - ○ ABRACADABRA
  - ○ BRACADABRAA
  - ○ RACADABRAAB
  - ○ ACADABRAABR
  - ○ CADABRAABRA
  - ○ ADABRAABRAC
  - ○ DABRAABRACA
  - ○ ABRAABRACAD
  - ○ BRAABRACADA
  - ○ RAABRACADAB
  - ○ AABRACADABR

- After Sorting
  - ○ AABRACADABR
  - ○ ABRAABRACAD
  - 2 → ○ ABRACADABRA
  - ○ ACADABRAABR
  - ○ ADABRAABRAC
  - ○ BRAABRACADA
  - ADABRAA
  - ○ CADABRAABRA
  - ○ DABRAABRACA
  - ○ RAABRACADAB
  - ○ RACADABRAAB

RDARCAAAABB

Downsides of Burrows-Wheeler Algorithm

- Have to process blocks of input file
    - Compare to LZW, which processes the input one character at time
- The larger the block size, the better the compression
    - But, the longer the sorting time

Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items
- Output:
  - repeatedly find a minimum item
- You are implementing an algorithm that repeats this problem
  - examples of such an algorithm?
    - Selection sort and Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

Let’s create an ADT!

- The Priority Queue ADT

- Let’s generalize min and max to highest **priority**
- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()
- We mentioned priority queues in building Huffman tries
- How do we implement these operations?
  - Simplest approach: arrays

Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - Θ(1)
- Find:
  - Search for the highest priority item (e.g., min or max)
  - Θ(n)
- Remove:
  - Search for the highest priority item and delete
  - Θ(n)

Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - Θ(n)
- Find:
  - Return the item at the end of the array
  - Θ(1)
- Remove:
  - Return and delete the item at the end of the array
  - Θ(1)

So what other options do we have?

- What about a balanced binary search tree?
    - Insert
        - Θ(lg n)
    - Find
        - Θ(lg n)
    - Remove
        - Θ(lg n)
- OK, all operations are Θ(lg n)
    - No constant time operations

Which implementation should we choose?

- Depends on the application
- We can compare the *amortized runtime* of each implementation
- Given a set of operations performed by the application:

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\#\text{operations}}$$

Example: Huffman Trie Construction

- K-1 iterations
  - K is the # unique characters in the file to be compressed
- Each iteration:
  - 2 removeMin calls
  - 1 insert call
- Unsorted Array: Total time Huffman Trie Construction =(K-1)*[2 * K + 1 * 1] = O(K^2)
- Sorted Array: Total time Huffman Trie Construction =(K-1)*[2 * 1 + 1 * K] = O(K^2)
- Balanced BST: Total time Huffman Trie Construction =(K-1)*[2 * log K + 1 * log K] = O(K log K)