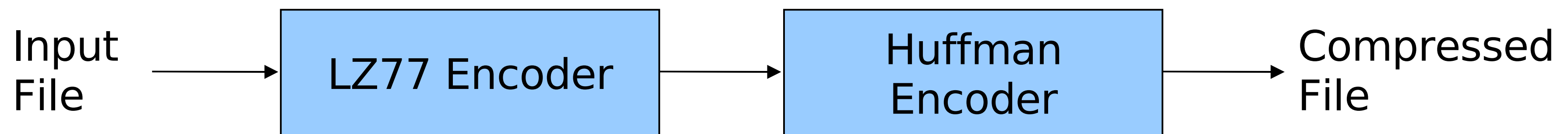
GZIP Encoding

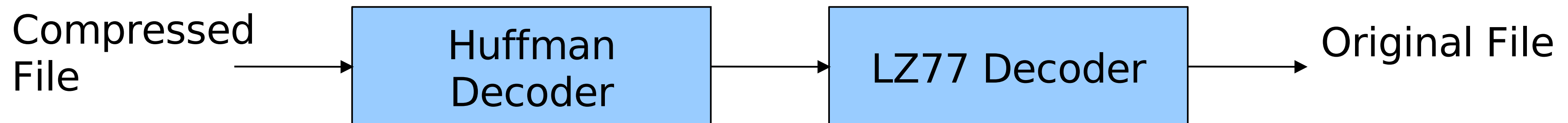
6.375 Final Project

Behram Mistree &
Dmitry Kashlev

GZIP - Outline

- GZIP
  - Lossless compression algorithm
  - Specified by RFC 1950, 1951, and 1952.
  - Two parts:
    - LZ77
    - Huffman Encoding





LZ77 – Basic Idea

LZ77 looks at partial strings within text.

- If a particular string occurred within the previous 32 Kb of data, replace it with a pointer to the previous string.
- Takes advantage of the repetitive nature of English text.

LZ77 - Example

Original Text:

This will be encoded.

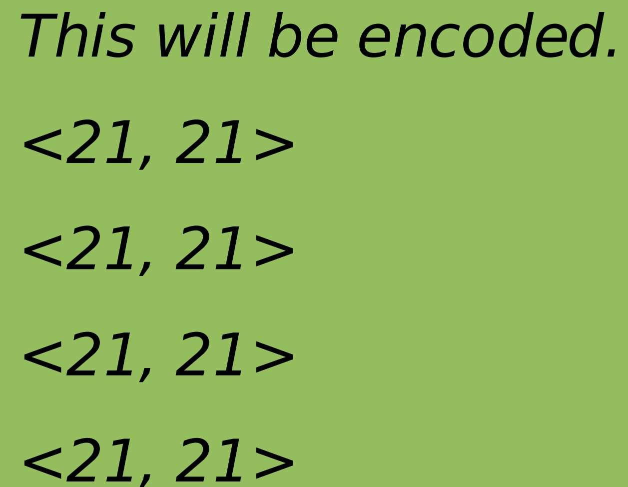
This will be encoded.

This will be encoded.

This will be encoded.

This will be encoded.

Encoded Text:



LZ77 - Encoder

- Decoupled 32Kb of memory
  - Stores last 32Kb of text file
- Memory manager
  - Writes to correct position of memory
  - Checks memory against input data
- Wrapper
  - Receives value from GET/PUT interface
  - Writes out either single character or encoded distance, length pair

LZ77 - Decoder

Deals with two cases:

- Case 1: Receives a character

  Pipes character directly to output.

- Case 2: Receives a length-distance pair

  Perform a memory lookup and write out string of characters to output buffer.

LZ77 - Clock Time and Area

- **Encoder*:**
  - 4051.00 μm^2
  - 3.93 ns critical path
- **Decoder*:**
  - 2015.25 μm^2
  - 3.93 ns critical path

* We could not get Encounter to synthesize memories correctly, so these values do not include a 32K long, 8 byte SRAM memory.

LZ77 - Initial Results Compression

Pre-Encoding: 108,673 character text file input.

Encoded: 27,052 pair and character values in encoded file.

- 23848 pair values.
- 3204 single characters slip through.
- Encoded gives 89,653 bytes*. 82% the size of the initial file without Huffman.

*Assuming a 29 bits for each pair.

LZ77 - Limiting Factors

- The encoding algorithm is the primary bottleneck.
- Relies on repetitive nature of document. In worst case (no repetitions), to encode each character will need to examine 32Kb of data.
- Algorithm is O(n) time. But it has a huge constant factor.

LZ77 - Exploration

- Memory requests can return more than one piece of data at a time.
  - Increase data from memory
- Increase concurrency
  - Can check for multiple characters and single characters concurrently

Huffman code

Every ascii character has an equivalent Huffman code

Huffman code is a sequence of bits.

The huffman sequences may have the same value, but different bit length

Example: 0011 and 11 are different huffman codes

Assuming the following alphabet:

D: 00
E: 11
H: 010
L: 011
O: 101
R: 1000
W: 1001
: 10001

HELLO WORLD

0101101101110110001100110110000
1100

Huffman Tree

Huffman code is a prefix-free code

Huffman code does not have a fixed length

During encoding, the ascii characters are generated by going down the binary Huffman tree starting at root node.

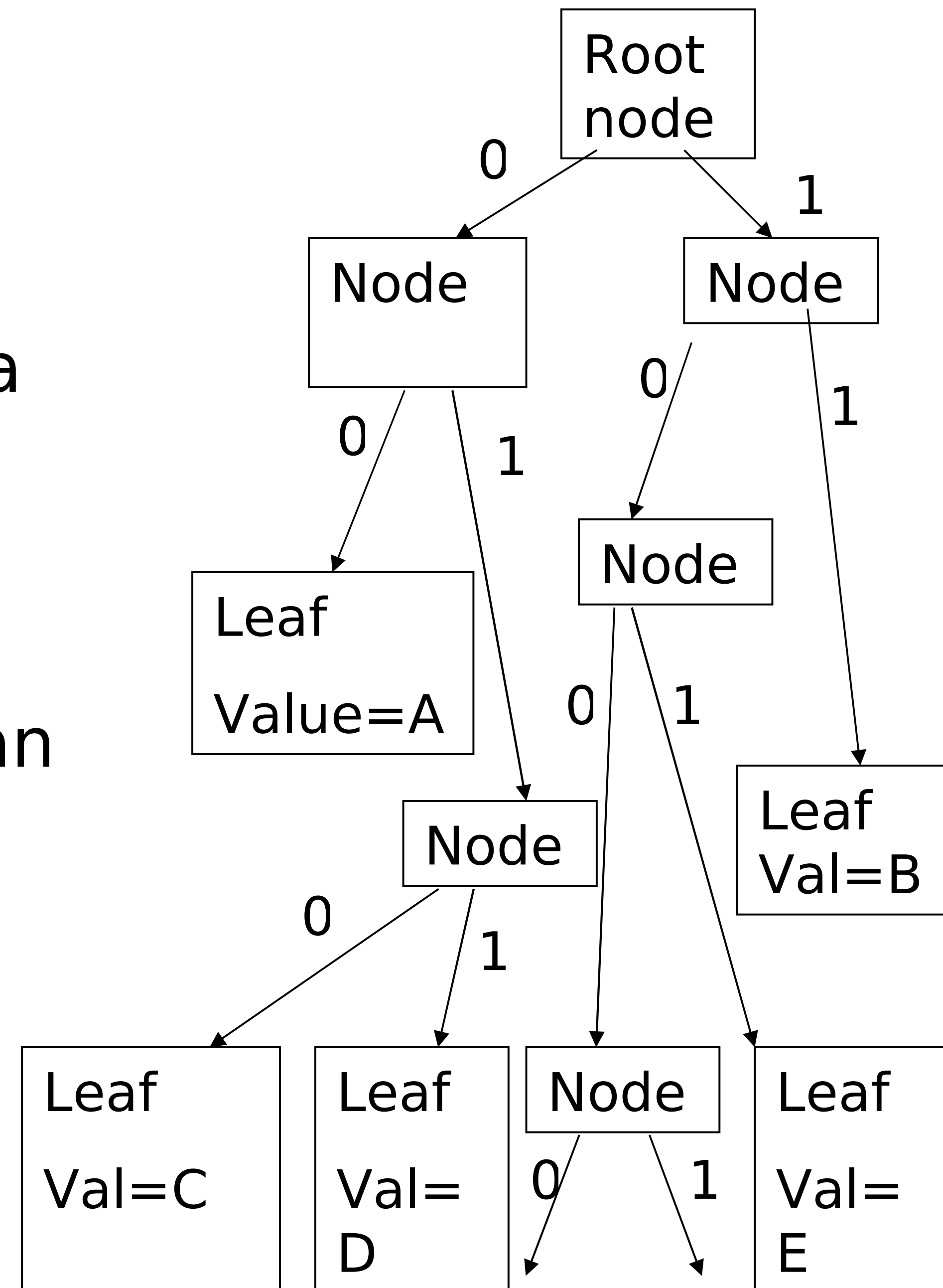
A: 00

B: 11

C: 010

D: 011

E: 101



Huffman Encoder

A Table mapping ascii characters to huffman code stored in a register file.

Register

index	Huffman code

Registers

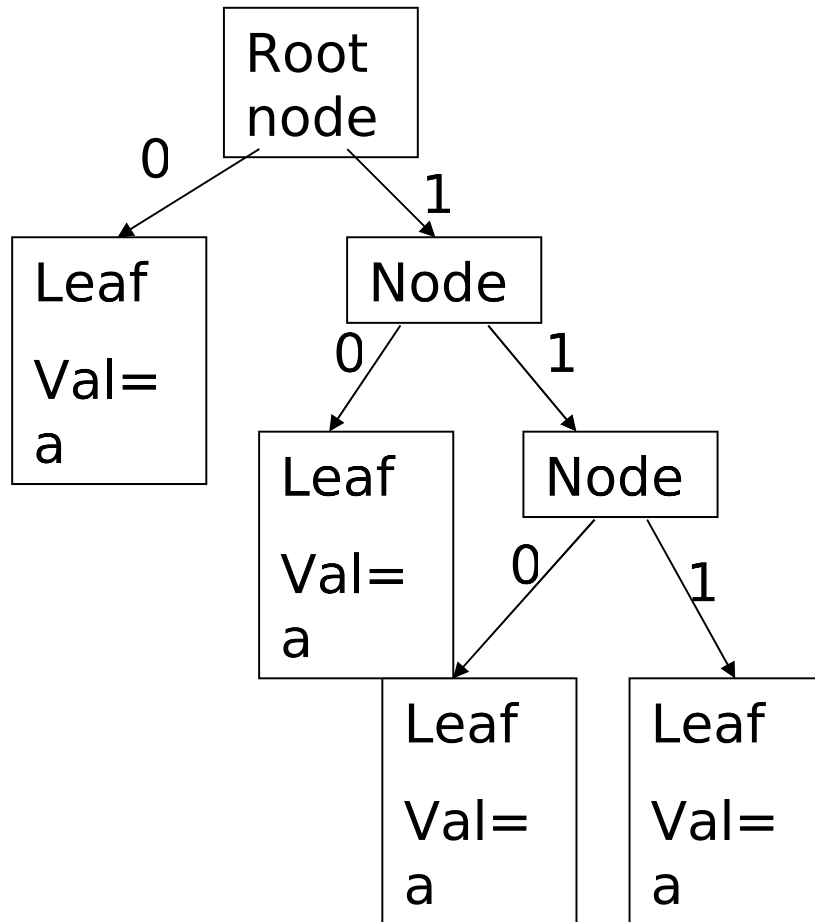
‘a’	97	1111111111111111....1111111111111110
‘b’	98	1111111111111111....11111111111111110
‘c’	99	1111111111111111....111111111111111110
‘d’	100	1111111111111111....1111111111111111110
‘e’	101	1111111111111111.... 11111111111111111110

For every ascii character in a file, perform table lookup to get huffman code for the character

Lookup is easy. Every register stores huffman code for equivalent ascii character (A=97 in ascii)

Huffman Decoder

A huffman tree is generated before decoding.



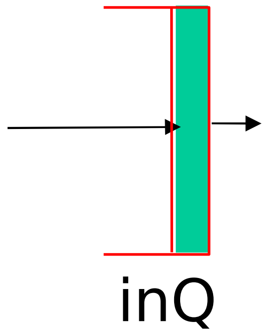
1	Root Node (leftPointer=2, rightPointer=3)
2	Leaf, (Value='a')
3	Node (leftPointer=4, rightPointer=5, Value=1)
4	Leaf (Value='b')
5	Node (leftPointer=6, rightPointer=7, Value=1)
6	Leaf (Value='c')
7	Leaf (Value='d')

Left pointer is taken if bit is “0”

Right pointer is taken if bit is “1”

Huffman module overview

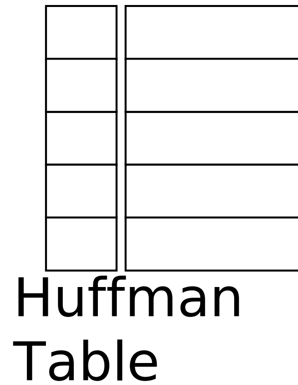
Encoder

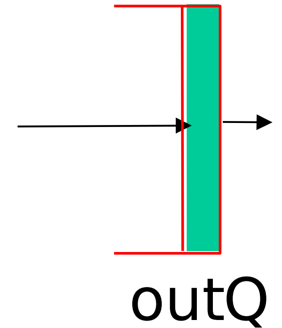


Input from LZ77 encoder CHAR or PAIR

If PAIR, pass to outQ

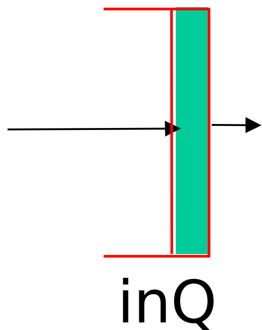
If CHAR, look up huffman code in table





Output to Huffman Encoder (1 bit)

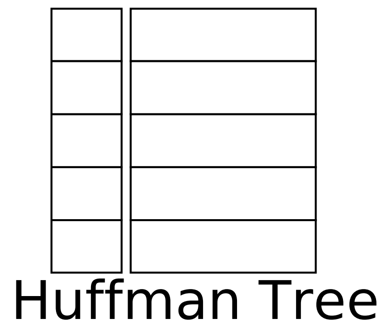
Decoder

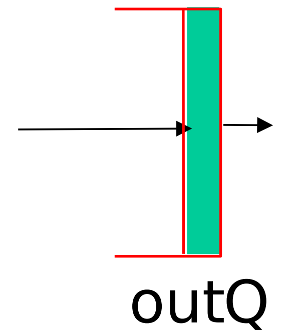


Input from Huffman Encoder (1 bit)

If PAIR, pass to outQ

If BIT, go down huffman tree to obtain ascii char





Output to LZ77 Decoder CHAR or PAIR

LZ77 and Huffman Results

Pre-Encoding: 108,673 character text file input.

Encoded: 161,177 bytes*.

*Assuming a 29 bits for each pair.

\0

Read new value
first[3] = infifo.first()
infifo.deq()

Initialization
Read 3 values into first[0], first[1], first[2] from infifo

check_equals
send request to mem check for all values of first

Case 1: Memory !contain first in consecutive order.

Write first[0]
Write first[0] into memory and to outfifo.

Shift
first[0] = first[1]
first[1] = first[2]

Case 2: Memory contains first in consecutive order.

look_for_more
writes first[0], first[1], and first[2] into memory
set count = 0.

Write
Write pair(count +3, distance) to outfifo. (note: distance is returned by check_equals)

Case 1: !=

check_equals
tmp = infifo.first()

Case 2: ==

write tmp to memory
infifo.deq()
++count

Things to add and improve

- Compress LZ77 pairs with huffman coding
- Dynamic Huffman
- Instead of large register file, use decoupled memory

Static vs. Dynamic Huffman

Two ways of generating huffman alphabet

2) Static Huffman – the huffman table and tree are generated before encoding/decoding takes place

3) Dynamic Huffman – The table is dynamically changing as new ascii characters are introduced, based on the frequency of ascii characters

LZ77 - Speed

- Required 2,854,403,040 clock cycles to perform encoding and decoding with LZ77 previous file.
- 11.2178039 s to perform both lz77 encoding and decoding
- 2,853,820,260 clock cycles to perform encoding and decoding with huffman and lz77.
- 11.2155136 s (assuming 3.93ns critical path).