Fabrice Kurmann
fkurmann@ucsc.edu
10 May, 2021

CSE 13S Spring 2021
Assignment 5: Huffman Coding
Design Document

Overview: In this assignment, a Huffman encoder is built. An input file is encoded into a compressed version of itself with the most common bytes being represented by bit strings smaller than a byte. The bit strings, codes, are assigned using a Huffman tree. The decoder then completes the reverse encoding process to output the original input without any loss in accuracy or detail.

Encoder: Find Huffman encodings if an input file, use those encodings to compress the file.

-h - print out program info message
-i - specify an infile other than stdin
-o - specify an outfile for the encoded codes only other than stdout.
-v - print compression statistics (uncompressed size, compressed size, space saving) to stderr

$$100 \times (1 - (\text{compressed} / \text{uncompressed})) = \text{saving}$$

Encoder Flow:

1) Parse command line args with getopt
2) Construct a histogram of 256 uint64ts
3) Increment the count of first and last histogram elements (0 and 255)
    This is to eliminate the errors associated with empty or single character infile.
4) Construct Huffman tree using a priority queue (build_tree)
    Create a priority queue with a Node for each symbol in the histogram with a positive frequency
    While there are two or more items in the priority queue, dequeue two of them, combine them to hame a parent node
        Parent node frequency = left + right child frequency
    The last node standing is the root.

5) Construct a code table, this is a simple 256 code array. (Build_codes)
    Starting at root traverse Huffman tree
        If current node is leaf, the code is the path from root to node
            Save the code to the code table
        Else the current node must be a branch, push 0 to the code and recursively apply the process to the left link.
        Pop a bit from the code, as with depth first search, then recurse down the right node. (Adding 1 to the code).
        Pop the most recent bit from the code.
6) Construct a header (ADT for tracking information that must be transferred from the input to the output.
7) Perform a post order traversal of the Huffman tree.
    Write L followed by the byte symbol for each leaf
    Write I followed by nothing for interior nodes.
8) For every item in the infile, use your code to write the corresponding code to the outfile. (Write_code)
    Flush remaining codes *to prevent data leaks?*

Decoder: Decode the Hamming codes made by the encoder.
Print statistics (bytes processed, uncorrected and corrected errors, error rate decimal to 6 digits of precision) to stderr.

Accept command line options:
-h - print out program info message
-i - specify an infile other than stdin
-o - specify an outfile for the decoded codes other than stdout.
-v - enables printing statistics to stderr, same stats as encode

Decoder Flow:

1) Parse command line options with getopt
2) If input magic number does not match macro magic number, end program
3) Read the input's Huffman tree
Store the tree in an array of size tree_size
Reconstruct the tree using rebuild_tree
Read over the tree array (tree_dump)
If element of the array is L, the next element is a leaf symbol, create a node for that leaf with node_create
Push the node to your stack
If element of the array is I, pop once for RIGHT child, again for LEFT child
Join children to make a parent node, push this onto the stack
Last element is the root of the Huffman tree.
4) Read the infile one bit at a time (read_bit)
If bit = 0, walk down left child of current tree node
If bit = 1, walk down right child of current tree node
If you hit a leaf node, write the leaf's symbol to the outfile and return back to the root to repeat.
5) Outfile is complete when it's length matches file size passed by the infile.

Reading and Writing with a BLOCK buffer:

Pseudocode:

```
read_bytes
    while (true)
        bytes_in = read (input_file, buffer, BLOCK)
        ** if less than a BLOCK is read, bytes in will be less than 4096, we then don't loop again.
        bytes_read += bytes in
        if (bytes_in < BLOCK)
            return bytes_read
            break;
```

Write bytes uses the same looping logic to write.

Write codes uses a Buffer and static buffer_index
if buffer_index = 4096 * 8, write bytes
clear buffer - this is essential to avoid printing junk values in your flush.

The Node ADT:

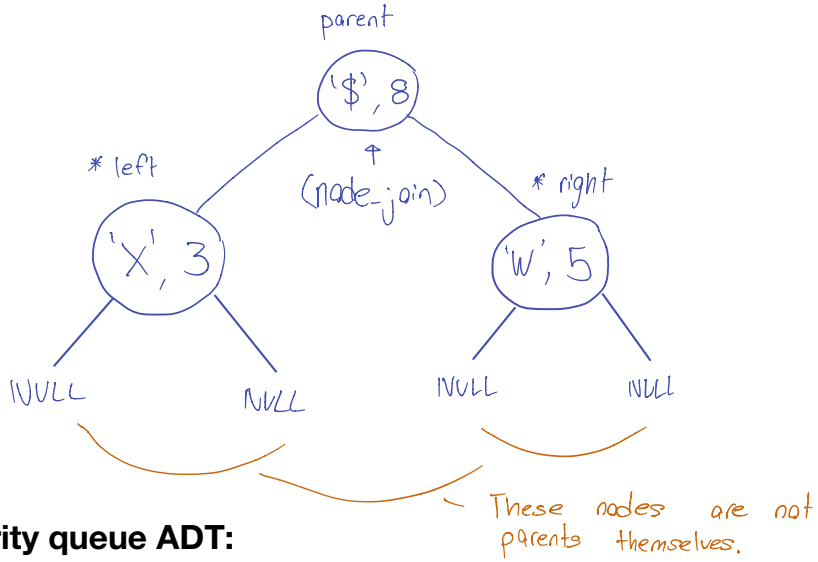
These are the base units Huffman trees are made of.

Nodes contain these variables:

Pointer to left child (Node pointer)
Pointer to right child (Node pointer)
It's own symbol (uint8)
The frequency of the symbol (used by encoder) (uint64)

Functions to implement:

node_create - create new node
node_delete - destructor, free the pointer
node_join - join a LEFT child to a RIGHT child, return a pointer to the created parent with the children set accordingly. Parent symbol = "$". Frequency = left frequency + right frequency
node_print - debug print function.



The priority queue ADT:

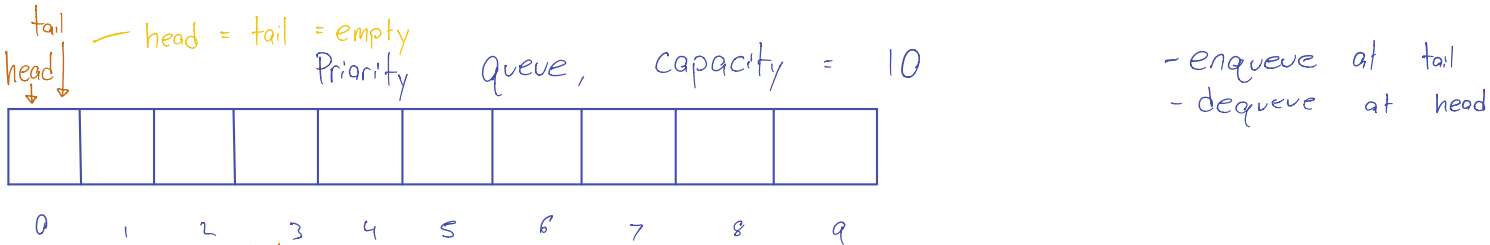
This priority queue stores nodes.

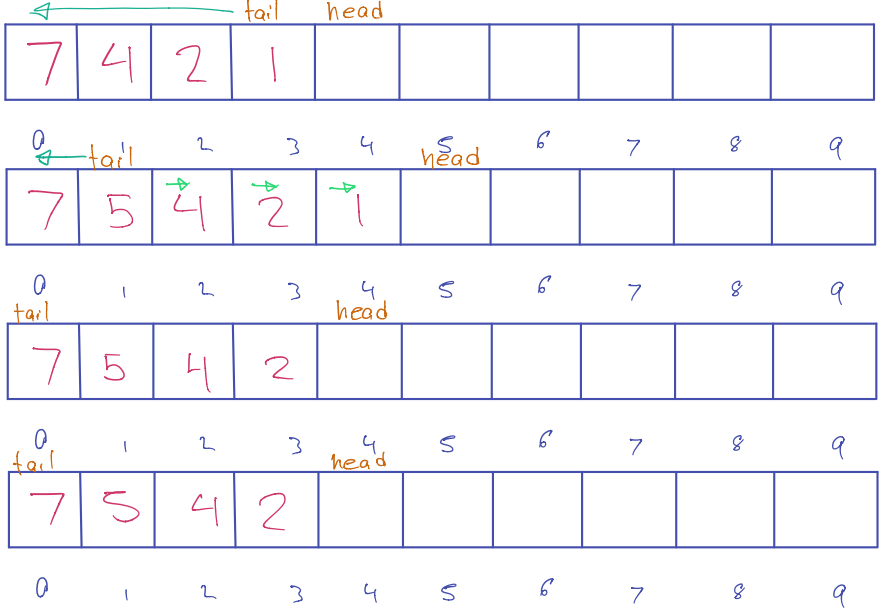
Like a queue but instead of first in first out, it is highest priority first out.

Priority is indicated by the frequency of a node (<u>lower frequency = higher priority</u>)

Prioridt queue functions:

pq_create - create a priority queue with a maximum size specified by input *capacity*
pq_delete - destructor, free the pointer
pq_empty - true if empty, false otherwise
pq_full - true if full, false otherwise
pq_size - return size
enqueue - return false if queue is full before enqueueing
dequeue - return false if queue is full before dequeuing
pq_print - display a tree of your ordered priority queue, *works best if queue is ordered by insertion sort*





enqueue(5)
- head moves
- tail goes to correct place
- inserts node to queue - all others shift
- returns to "bottom or end"

dequeue()
- head moves
- dequeu what is at head

size()
return (head)
↳ since tail always @ 0.

Note, because of the need to incorporate insertion in the priority queue regardless, I am able to prevent the circular queue phenomenon where the tail follows the head leading to weird edge cases regarding looping around the queue. The tail is always going to return to index 0, it will then insert, shift all items over, and spring back. With every insertion, the head moves one index up, with every dequeue, it moves on index down. If the head = tail (something that can ONLY happen at index 0) the queue is empty.

The Code ADT:

This is a stack of bits that is responsible for creating a code for each symbol in the Huffman tree.

Code functions:

code_init - NO dynamic memory allocation so no destructor needed. To initialize, zero out the array of bits: *bits*. Set top to 0.

code_size - return size, equal to the number of bits pushed.

code_empty - return true if stack empty, false otherwise

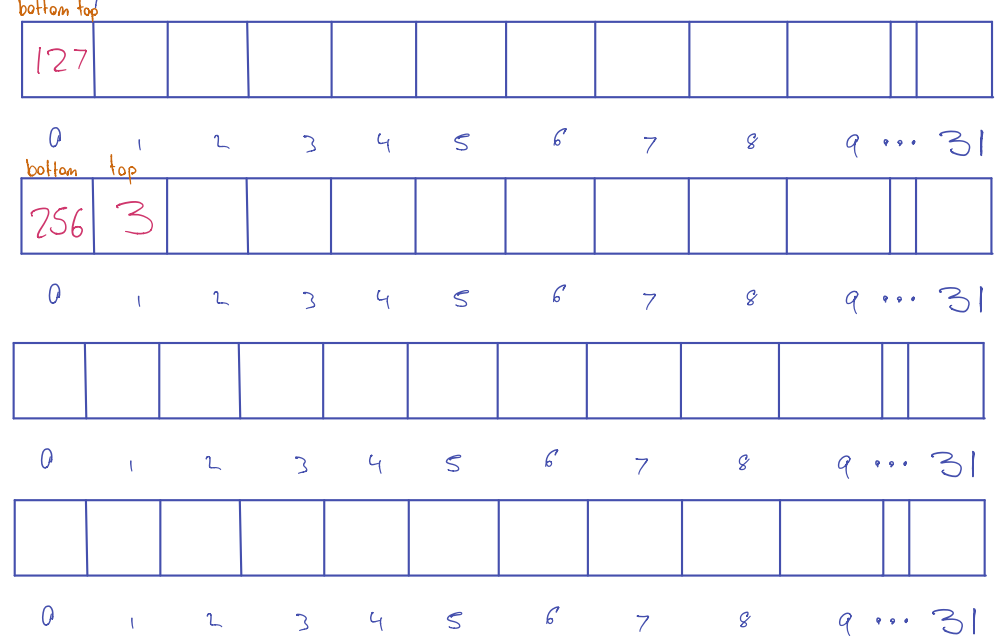
code_full - return true of stack full, false otherwise

code_push_bit - push bit, or return false if code is full

code_pop_bit - pop bit to pointer, or return false if code is empty

code_print - print code stack

Array for code stack - 32 bytes = 256 bits long - always - not dynamic



push_bit 7x → byte 1: 01111111
= 1+2+4+8+16+32+64 = 127

push-bit 3x → byte 1: full = 256
→ byte 2 push 2 0...11=3
- add bits until byte is full
- move top, add rest of bits

empty() - top == bottom AND byte(bottom) = empty

Pseudocode for code stack functions:

code_full()
    if code->top == code_length AND code->items(top) == 256 - aka, if all bits are full
        return true
    else return false

code_empty()
    if code->top == code_botton AND code->items(bottom) == 0
        return true
    else return false

code_push()
    if code_full == false
        if code->items(top) == 256 - aka current byte is full
            top++
        code->items(top) << - aka shift all bits one to the left
        code-> items(top) ++ - add 1, so that all bits (with one additional one) are now set.
    else return false

code_pop()
    if code_empty == false
        if code->items(top) == 0 - aka current byte is already empty
            top- -
        code->items(top) >> - shift right (you could also just do integer division by 2)
    else return false

The Stack of Nodes ADT:

Same stack as from asgn3, 4 however, instead of storing an array of items, this stack stores nodes of items.

Effected stack functions:

stack_push - push a node onto the stack, or return false if stack is full

stack_pop - pop a node from the stack to the pointer, or retur false if stack is full

The Header ADT:

Header is simply a storage structure which contains information (metadata?) on the infile, the information on the Huffman tree the decoder will need to replicate, and the file size to aim for when decoding.

I/O functions:

read_bytes - read bytes from an infile into a buffer, stop when the infile is parsed OR you have read the specified number of bytes - using a loop

write_bytes - write a specified number of bytes (or all bytes) from a buffer into the outfile using a loop

read_bit - places bytes into a static buffer of bytes - using an index to track the bit you are on, read every individual bit of the bytes in the buffer - refilling the buffer when it has been read until your infile is finished.

write_code - set single bits from the code ADT to the buffer, when the buffer is full, send the contents to the outfile

flush_codes - A function to clear bits remaining in the buffer after encoding is complete, *clear bits in last input byte before flushing***

Read/Write codes - use bitwise operations for printing single bits:

* accessing specific bit

$= 00110100_2 = 4 + 16 + 32 = 52 =$ uint value

To access bit 3: 52 → USE Logical shifts until 3 is bit 0
- if uint = odd : value = 1
  else : value = 0

Huffman.h functions:

build_tree - given a computed histogram construct a Huffman tree.
Histogram will have 256 indices, one for each symbol
Return the root node for your Huffman tree

build_codes - fill the code table with code for each of the symbols in the Huffman tree
constructed codes are copied to a CODE ADT table with 256 bitwise indices

rebuild_tree - using the tree_dump array, reconstruct the same Huffman tree.
Return root node for the reconstructed tree

delete_tree - destructor for a tree, this must free ALL nodes and set tree pointer to Null.

Example of the encoding/tree making process:

First create a histogram: Word to encode:
Doppelkupplungsgetriebe 23 = length, 14 different elements

Histogram:

D	o	p	e	l	k	U	n	g	s	r	i	b	t
1	1	4	4	2	1	2	1	2	1	1	1	1	1

PQ → Priority queue for this histogram

[P,4 e,4 l,2 U,2 g,2 D,1 o,1 k,1 b,1 n,1 s,1 r,1 i,1 t,1]

Nodes for this histogram:

D,1 o,1 P,4 e,4 l,2 k,1 b,1

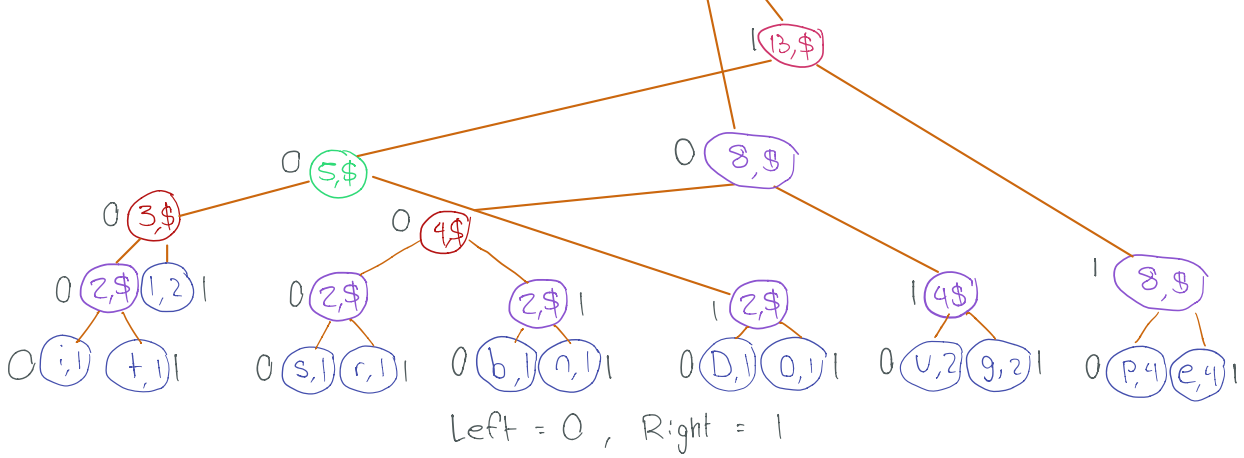
U,2 n,1 g,2 s,1 r,1 i,1 t,1

Constructing the Huffman Tree:

PQ

[ROOT 21,$ 13,$ 8,$ 8,$ 5$ 4$ 3$ P,4 e,4 3$ 2$ 2$ 2$ 2$ l,2 U,2 g,2 D,1 o,1 k,1 b,1 n,1 s,1 r,1 i,1 t,1]

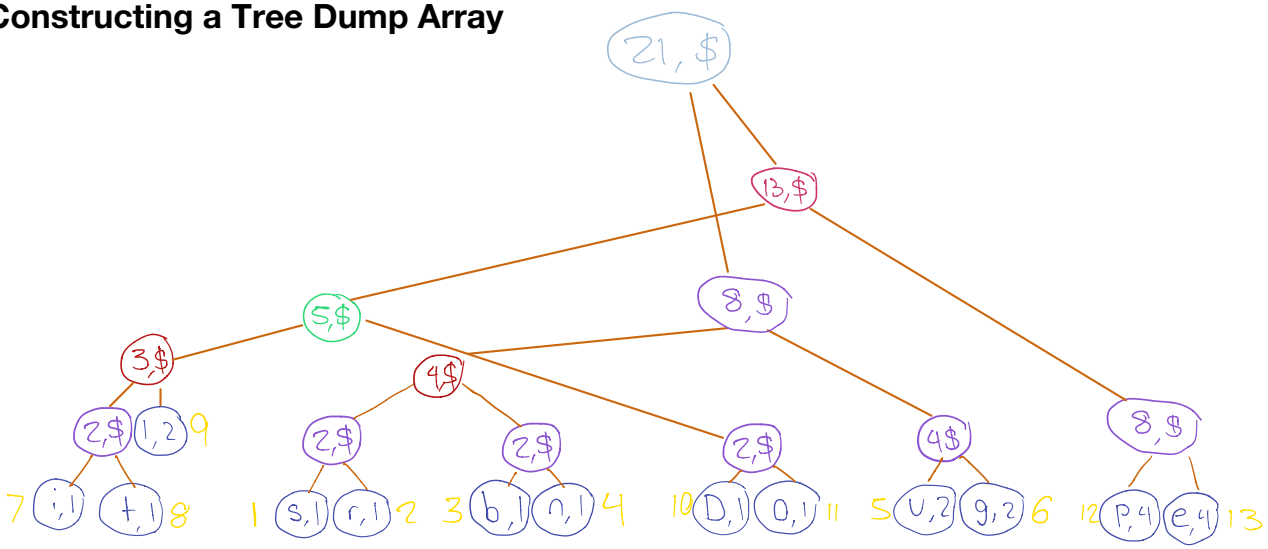
Dequeue 2, enqueue parent - until one node left



Constructing a code table

D	1010
O	1011
P	110
e	111
L	1001
t	1000 1
U	010
n	0011
g	011
s	0000
r	0001
i	10000
b	0010

Constructing a Tree Dump Array



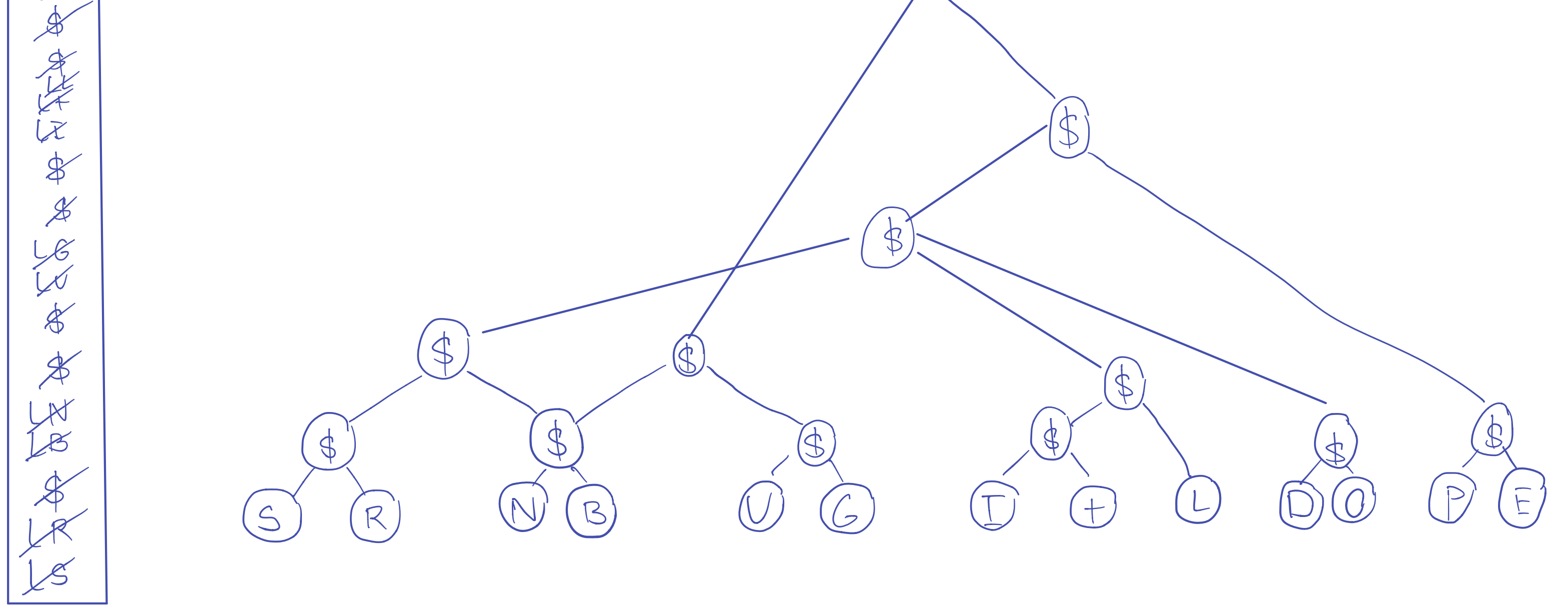
Tree Dump :

[Ls, Lr, I, Lb, Ln, I, I, LU, Lg, I, I, Li, Lt, I, LL, I, LD, LO, I, I, LP, Le, I, I, I]

Reconstructing a tree from the tree dump array (using a stack)

Stack for reconstruction:

Reconstructed Tree :

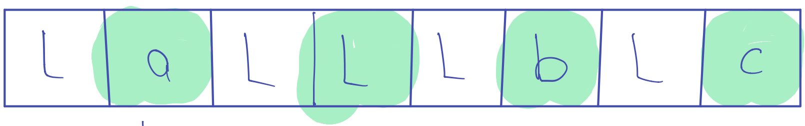


Handling unique character cases:

When rebuilding the Huffman tree, the character L is used to indicate a leaf node, meanwhile the character I is used to indicate an interior node. Both these characters may also be actual node values, so to prevent reconstructing an incorrect tree, the program must be able to tell a leaf or interior indicator from a node which simply has a character I or L.

Tree Dump

'L' scenario:

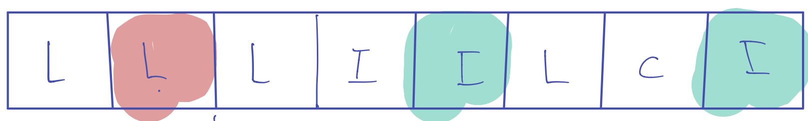


- green = items to push

L, push a
a, pass
L, push L
L - despite L, don't push
L, push b
b, pass
L, push c

if L
- if before != L
  push
- if before == L AND 2before == L
  push

'I' scenario:



- red = L to push
- green = items to pot

if before != L
  push

if before == L AND 2before == L
  push