



# Greedy algorithms 2

## Caching, Compression

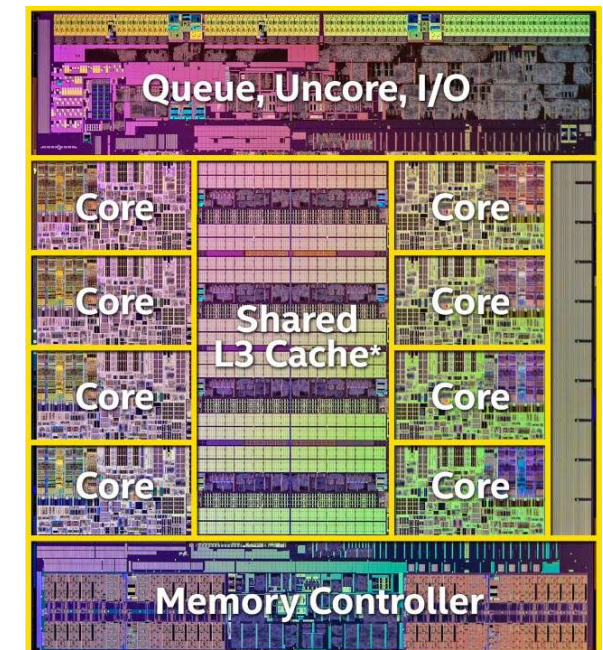
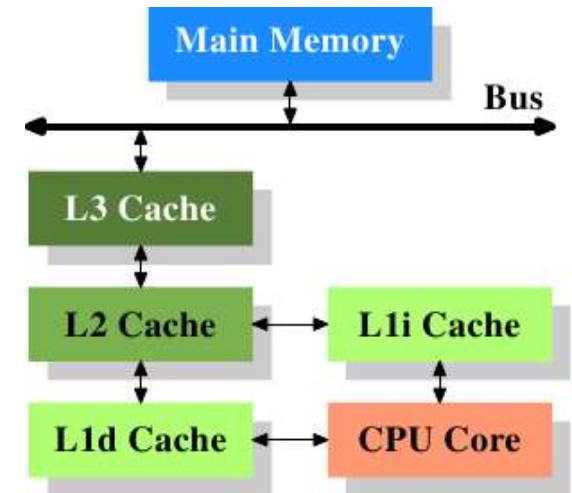
CS240

Spring 2022

*Rui Fan*

# Caching

- Cache is a piece of on-chip (fast) memory.
- Reduces effective access time to slow main memory.
  - When accessing memory, first try to find it in cache.
  - Only if it's not in cache do we access main memory.
  - Typically cache has ~1-20 cycles latency, memory has ~200-500 cycles latency.
- Since caches are on-chip, amount is usually quite small (~32 KB to ~4 MB).
  - Only store the most important, frequently accessed data in cache.
- Use caching algorithm to select which data to store.



# Optimal Offline Caching

## Caching.

- Cache with capacity to store  $k$  items.
- Sequence of  $m$  item requests  $d_1, d_2, \dots, d_m$ .
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must load requested item from main memory into cache, and evict some existing item, if full.

**Goal.** Eviction schedule that minimizes number of evictions.

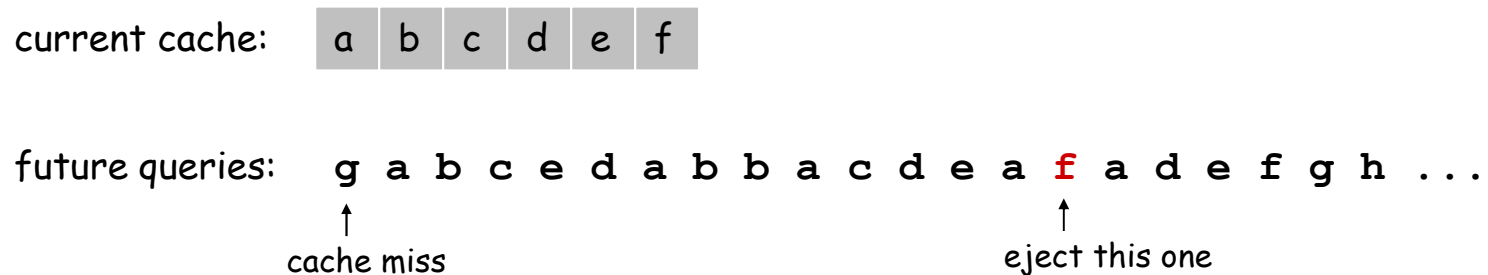
**Ex:**  $k = 2$ , initial cache =  $ab$ ,  
requests:  $a, b, c, b, c, a, a, b$ .

**Optimal eviction schedule:** 2 evictions.

a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b
requests	cache	

# Optimal Offline Caching: Farthest-In-Future

**Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future.



**Theorem.** [Bellady, 1960s] FF is optimal eviction schedule.

**Pf.** Algorithm and theorem are intuitive; proof is subtle.

# Reduced Eviction Schedules

**Def.** A **reduced** schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

**Intuition.** Can transform an unreduced schedule into a reduced one with no more evictions.

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	x	b
b	a	c	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

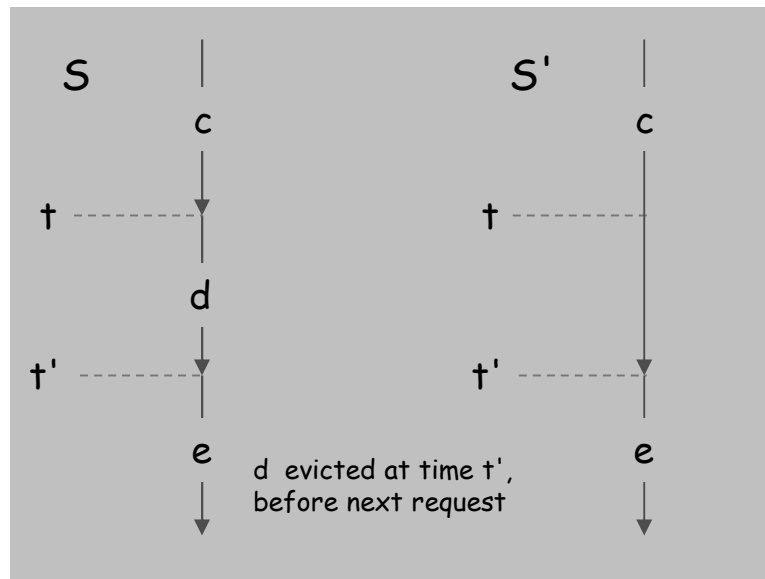
a reduced schedule

# Reduced Eviction Schedules

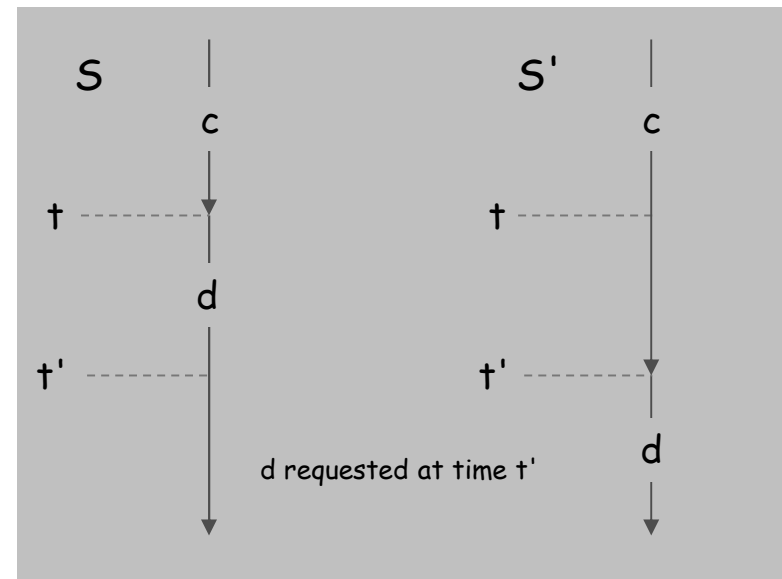
**Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more evictions.

**Pf.** (by induction on number of unreduced items) ← doesn't enter cache at requested time

- Suppose  $S$  brings  $d$  into the cache at time  $t$ , without a request.
- Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache.
- Case 1:  $d$  evicted at time  $t'$ , before next request for  $d$ .
  - $S'$  has one less eviction than  $S$  by time  $t'$ .
- Case 2:  $d$  requested at time  $t'$  before  $d$  is evicted.
  - $S'$  and  $S$  have same number of evictions by time  $t'$ .



Case 1



Case 2

# Farthest-In-Future: Analysis

**Lemma.** Let  $S$  be a reduced schedule that makes the same schedule as  $S_{FF}$  through the first  $j$  requests. Then there is a reduced schedule  $S'$  that makes the same schedule as  $S_{FF}$  through the first  $j+1$  requests, and incurs no more evictions than  $S$  does.

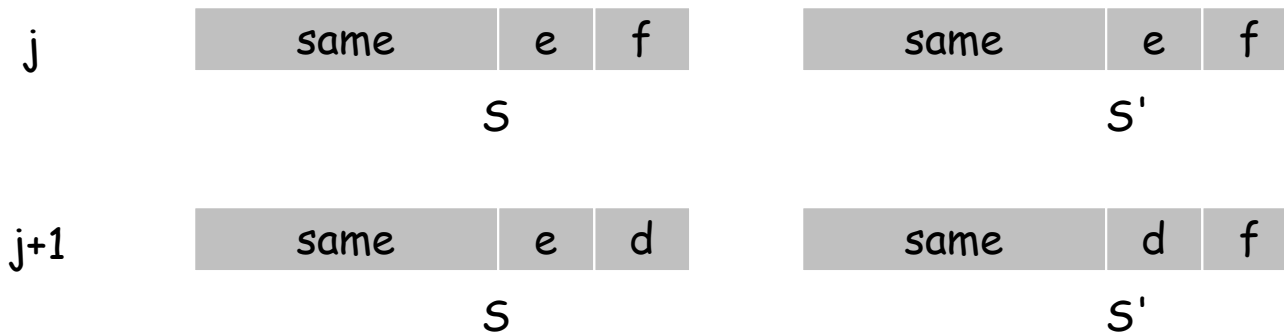
**Pf.**

- Consider  $(j+1)^{st}$  request  $d = d_{j+1}$ .
- Since  $S$  and  $S_{FF}$  have agreed up until now, they have the same cache contents before request  $j+1$ .
- Case 1: ( $d$  is already in the cache).  $S' = S$
- Case 2: ( $d$  is not in the cache and  $S$  and  $S_{FF}$  evict the same element).  
 $S' = S$

# Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: ( $d$  is not in the cache;  $S_{FF}$  evicts  $e$ ;  $S$  evicts  $f \neq e$ ).
  - begin construction of  $S'$  from  $S$  by evicting  $e$  instead of  $f$



- now  $S'$  agrees with  $S_{FF}$  on first  $j+1$  requests
- From request  $j+2$  onward, we make  $S'$  the same as  $S$ , but this becomes impossible when  $e$  or  $f$  is involved



# Farthest-In-Future: Analysis

Pf. (continued)

Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$ .

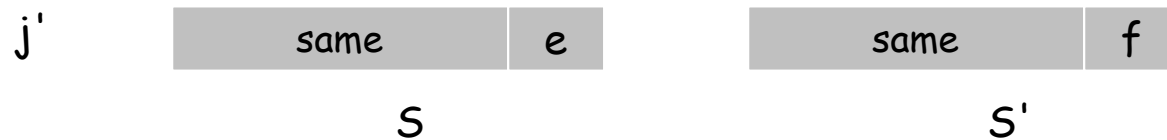


- Case 3a:  $g = e$ .
  - $f$  couldn't have been requested between time  $j+1$  and  $j'$ , because if it had been,  $S$  and  $S'$  would have taken different actions before  $j'$ .
  - So  $e$  is requested before  $f$ . But this can't happen with Farthest-In-Future, since  $S_{FF}$  evicted  $e$ , implying  $f$  is requested before  $e$ .

# Farthest-In-Future: Analysis

Pf. (continued)

Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$ .

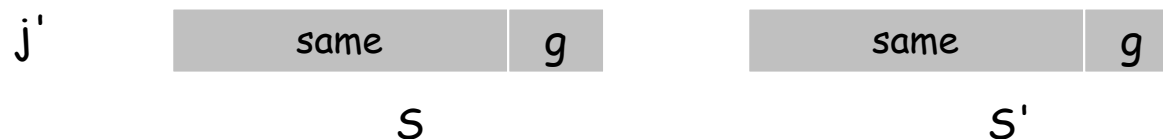


$S$  can't evict  $f$ , and if it evicts  $e' \neq e, f$ , then  $S'$ , by construction, would do the same thing.



- Case 3b:  $g \neq e, f$ .  $S$  must evict  $e$ .

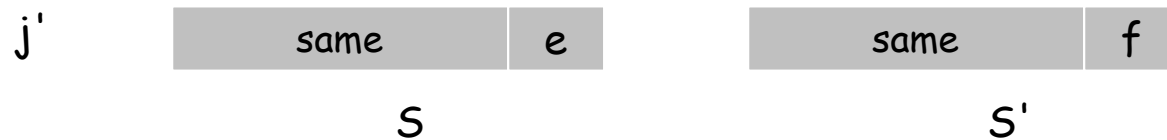
Make  $S'$  evict  $f$ ; now  $S$  and  $S'$  have the same cache. ▪



# Farthest-In-Future: Analysis

Pf. (continued)

Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$ .



- Case 3c:  $g = f$ . Element  $f$  can't be in cache of  $S$ , so let  $e'$  be the element that  $S$  evicts.
  - if  $e' = e$ ,  $S'$  accesses  $f$  from cache; now  $S$  and  $S'$  have same cache
  - if  $e' \neq e$ ,  $S'$  evicts  $e'$  and brings  $e$  into the cache; now  $S$  and  $S'$  have the same cache.
  - $S'$  is no longer reduced, but can be transformed using procedure on slide 6 into a reduced schedule which
    - a) agrees with  $S_{FF}$  through step  $j+1$
    - b) has no more evictions than  $S$

# Farthest-In-Future: Analysis

**Theorem.** FF is optimal eviction algorithm.

**Pf.** (by induction on number of requests  $j$ )

Base case (trivial):

There exists an optimal reduced schedule  $S$  that makes the same schedule as  $S_{FF}$  through the first  $0$  requests.

Inductive step (implied by the lemma):

If there exists an optimal reduced schedule  $S$  that agrees with  $S_{FF}$  through the first  $j$  requests,

then there exists an optimal reduced schedule  $S'$  that agrees with  $S_{FF}$  through the first  $j+1$  requests

# Caching Perspective

## Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

**LIFO.** Evict page brought in most recently.

**LRU.** Evict page whose most recent access was earliest.

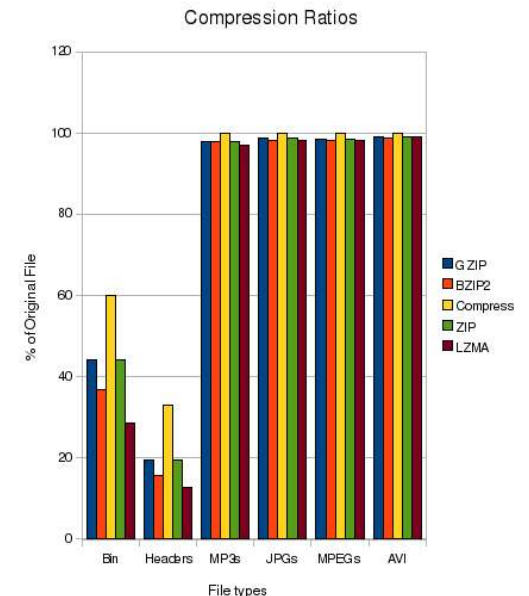
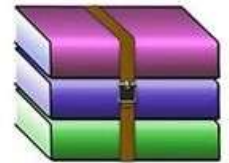
↑  
FF with direction of time reversed!

**Theorem.** FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is  $k$ -competitive.
  - I.e. it does  $\leq k$  times more loads than the optimal eviction algorithm.
- LIFO is arbitrarily bad.
  - For  $n$  requests, it may do  $O(n)$  times more loads than optimal.

# Compression

- Storing and transmitting data is expensive. Compression represents data more compactly.
- ASCII has 256 characters, so we use  $\log_2(256)=8$  bits to represent each character.
- But typically some characters **appear more often** than others. So we shouldn't use same number of bits for all letters.
- Basic idea for compression is to use different length bitstrings.
  - Use **short bitstrings** to represent **common characters**.
  - Use **long bitstrings** to represent **uncommon characters**.
  - We save space on average.





# Lossless vs. lossy compression

- Different algorithms for different applications.
- **Lossless** compression used in settings where losing even one bit can make data useless.
  - **Ex** Computer code, financial document.
  - Typical compression ratio is 2:1.
  - Huffman encoding is lossless.
- **Lossy** compression used when data still useful after losing some information.
  - **Ex** Audio and video, MP3 and MPEG.
  - We can't hear high frequencies or see fast movement, so we can discard this info.
  - Typical compression ratio is 5:1-50:1.



# Variable length encoding

- Let's compress “**lollapalooza**”.
- There are 5 different letters. If we use the same length bitstring to represent each letter, we need 3 bits per letter.
  - We use 36 bits total.
- To use different length bitstrings, first count how many times each letter appears.
  - 4 l's, 3 a's, 3 o's, 1 p, 1z.
- Use shorter bitstrings for more frequent letters.
- Use the encoding **l=00, a=01, o=10, p=110, z=111**.
  - Encoding of “lollapalooza” is **00100000011100100101011101**, formed by replacing each letter by its encoding.
  - We use 26 bits, for a 28% savings.





# Ambiguity

- We want the codewords to be short, but we also need them to be **unambiguously decodable**.
- **Ex** If we use l=0, a=01, o=10, p=110, z=111, then “lollapalooza” is only 22 bits.
  - But we can’t decode this encoding!
  - If we see 00010, we can’t tell whether this is encoding lal=[0,0,01,0], or llo=[0,0,0,10].
- We could use a separator, 0#0#01#0 vs 0#0#0#10. But that’s wasteful.
- Instead, we use **prefix-free codes**, which are unambiguously decodable.



# Prefix-free codes

- Let  $W$  be the set of codewords we use. Then  $W$  is prefix-free if no codeword in  $W$  is a prefix of another codeword.
  - 00, 10, 001, 100 is not prefix-free.
    - 00 is a prefix of 001, and 10 is a prefix of 100.
  - 00, 01, 10, 110, 111 is prefix-free.
- Prefix-free codes allow **unique decoding**.
  - Given the encoded string, just keep reading until you've read a complete codeword.
  - This codeword can't be part of a longer codeword, because the code is prefix-free.

# Decoding prefix-free codes

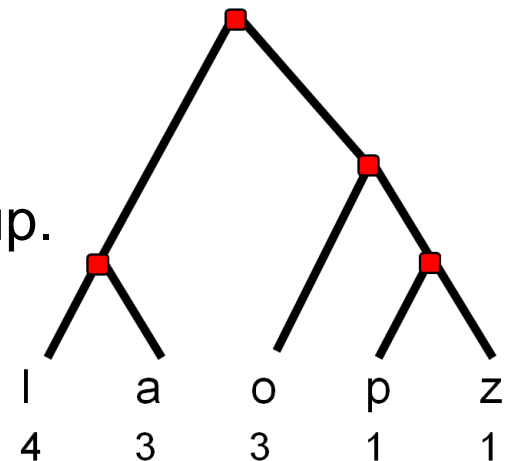
- Let  $S=00100000011100100101011101$ ,  
 $W=\{00,01,10,110,111\}$ , representing  
l,a,o,p,z.

00100000011100100101011101	00→l
00100000011100100101011101	10→o
00100000011100100101011101	00→l
00100000011100100101011101	00→l
00100000011100100101011101	00→a
00100000011100100101011101	110→p

...

# Huffman coding

- Huffman encoding is an optimal prefix-free code, invented in 1951.
- First, find the frequencies of the letters in your text.
  - For “lollapalooza”, it’s [l,a,o,p,z] → [4,3,3,1,1].
- Now, build a binary tree on the letters bottom up.
  - Make each letter a leaf, and set its weight to its frequency.
  - Take the **two lowest weight** nodes
    - Make them the children of a parent node.
    - Set the weight of the parent node equal to the sum of the weights of the two children.
    - Remove the two nodes.
    - Notice this is a **greedy** step.
  - Repeat till all nodes part of one tree.
- Represent left by 0, right by 1.
  - A letter’s encoding is represented by its **path from the root**.





# Huffman coding example

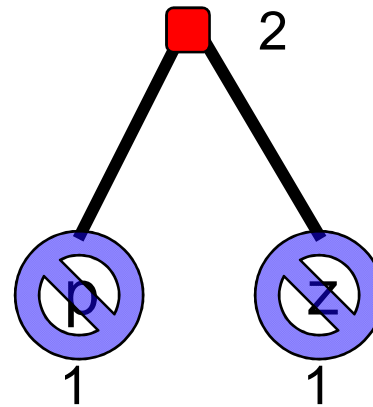
l	a	o	p	z
4	3	3	1	1

# Huffman coding example

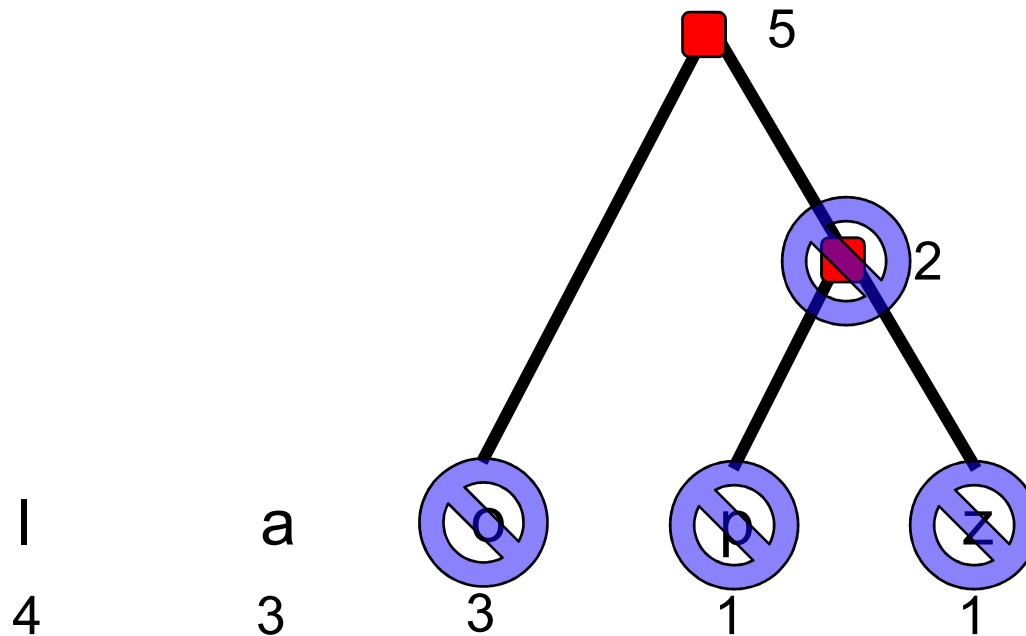
l  
4

a  
3

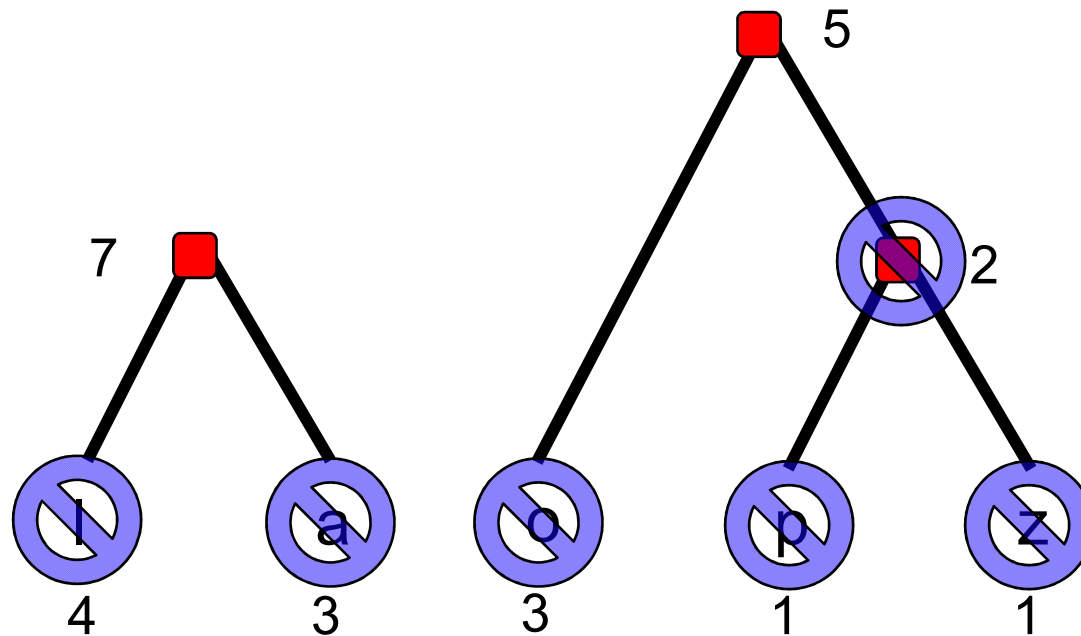
o  
3



# Huffman coding example

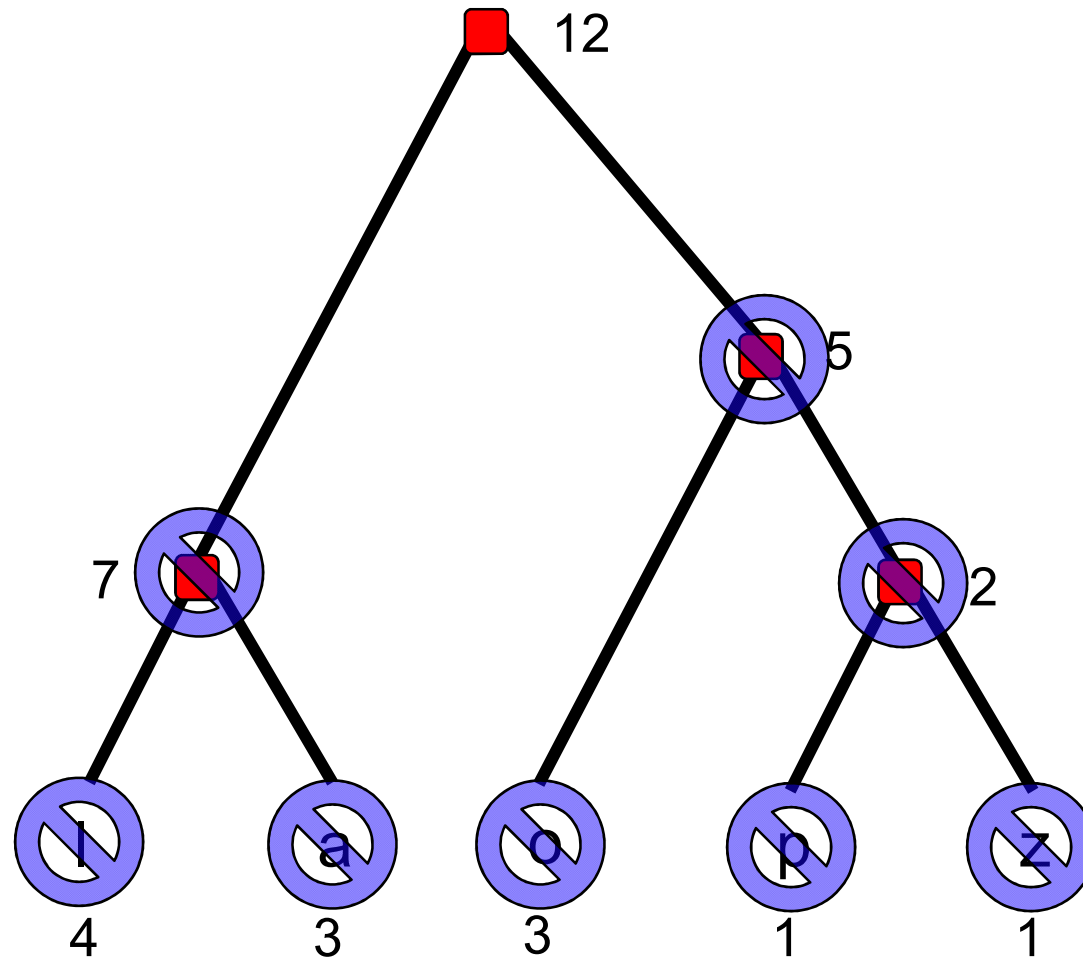


# Huffman coding example

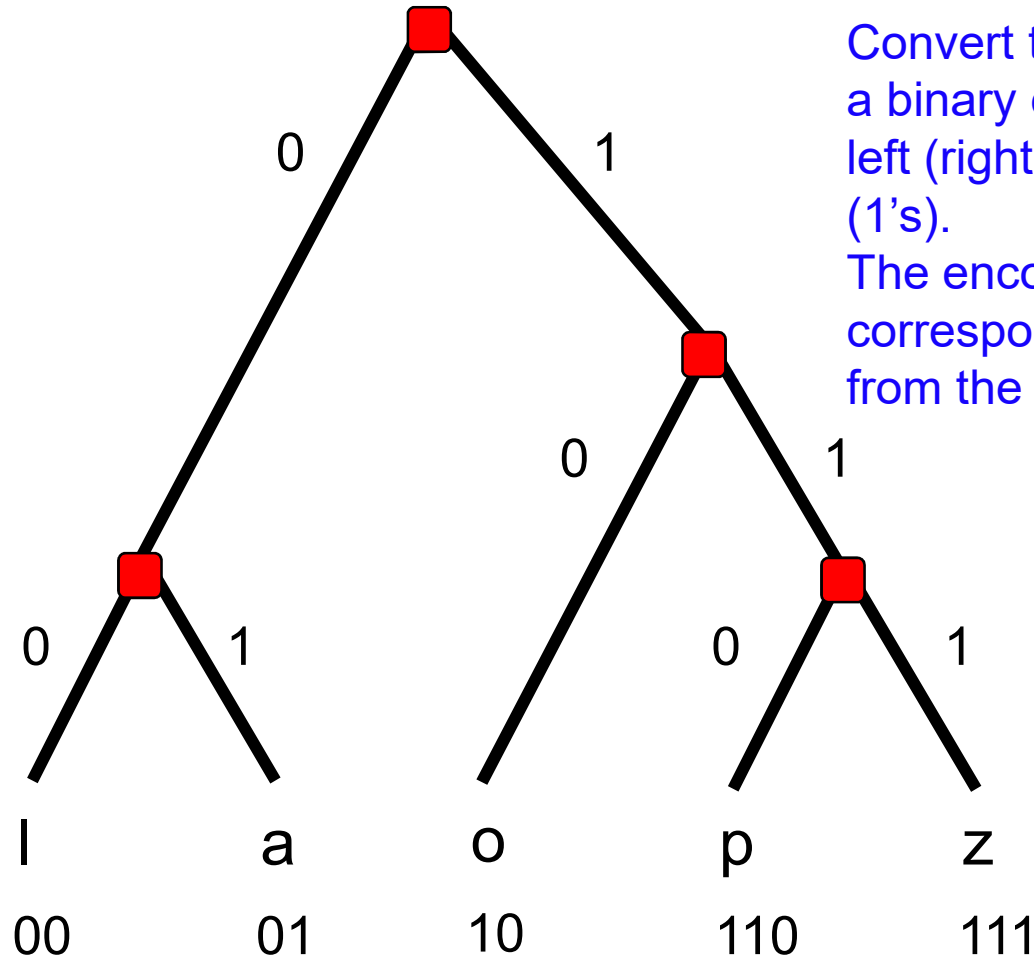




# Huffman coding example



# Huffman coding example



Convert the Huffman tree into a binary encoding, by treating left (right) branches as 0's (1's).  
The encoding of the letters corresponds to their path from the root.



# Huffman implementation

Let  $S=s_1s_2\dots s_n$  be a string. Let  $f(s)$  be the number of occurrences of char  $s$  in  $S$ .

for  $i=1$  to  $n$

    add  $(s_i, f(s_i))$  to a min-heap  $H$

for  $i=1$  to  $n-1$

    left  $\leftarrow$  removeMin( $H$ )

    right  $\leftarrow$  removeMin( $H$ )

    make a new node parent

    set parent's left and right children to left, right

    add(parent,  $f(\text{left}) + f(\text{right})$ ) to  $H$

a letter's encoding is represented by its path

# Huffman's complexity

Total time is  $O(n^2)$ .

for  $i=1$  to  $n$

add  $(s_i, f(s_i))$  to a min-heap  $H$

Each add takes  $O(\log n)$  time.

for  $i=1$  to  $n-1$

left  $\leftarrow$  removeMin( $H$ )

Each remove takes  $O(\log n)$  time.

right  $\leftarrow$  removeMin( $H$ )

make a new node parent

set parent's left and right children to left, right

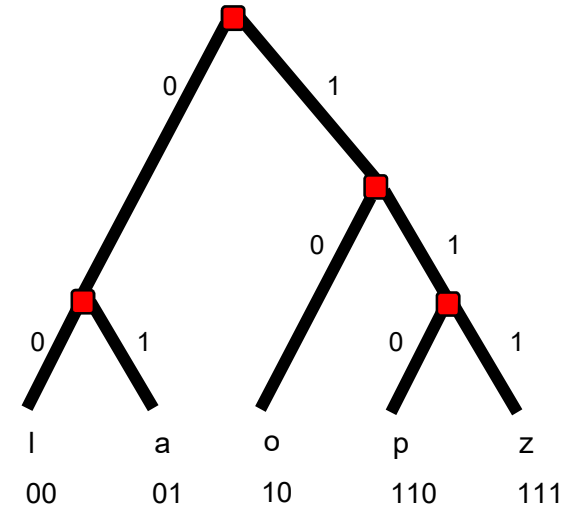
add(parent,  $f(\text{left}) + f(\text{right})$ ) to  $H$

There are  $n$  letter, each has  $O(n)$  height, so the total time is  $O(n^2)$ .

a letter's encoding is represented by its path.

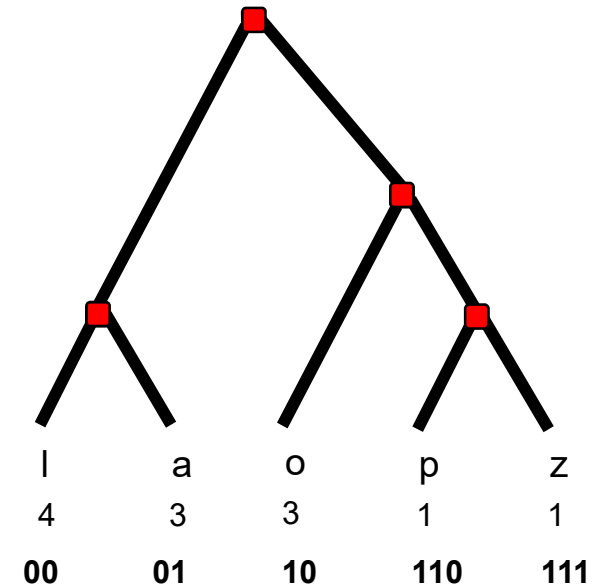
# Huffman code is prefix-free

- Any two codewords correspond to paths from the root to leaves.
  - The 2 paths split from each other somewhere.
  - After the split, neither codeword is a prefix of the other.
- Huffman codes are uniquely decodable.



# Huffman code is optimal

- Huffman encoding gives the **shortest uniform encoding** of strings.
  - Uniform basically means you can't change your encoding method for different strings.
- Call an encoding in which codewords are derived from paths in trees a **tree code**.
- **Fact** There exists tree codes that are optimal.
- Since the Huffman code is a tree code, to prove Huffman is optimal, we just need to prove it's an optimal tree code.

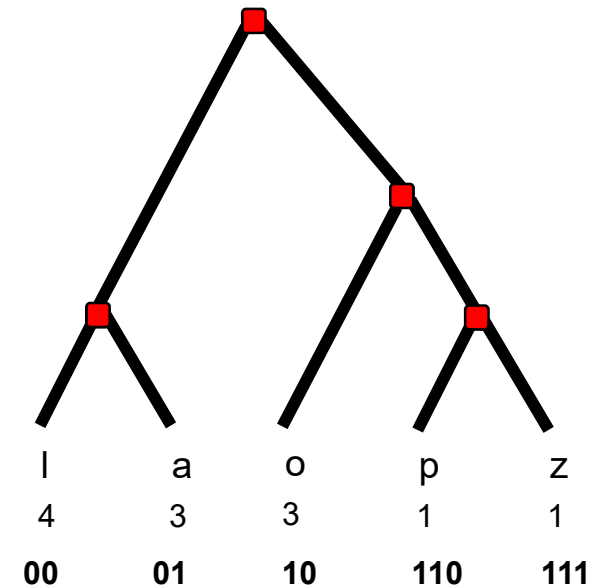


# Huffman code is optimal

- **Def** The cost of a tree code is  $cost(T) = \sum_{v \in T} d(v) \cdot f(v)$ , where  $d(v)$  denotes the depth of letter  $v$ , and  $f(v)$  denotes its frequency.

□  $cost(T)$  = number of bits to represent original string.

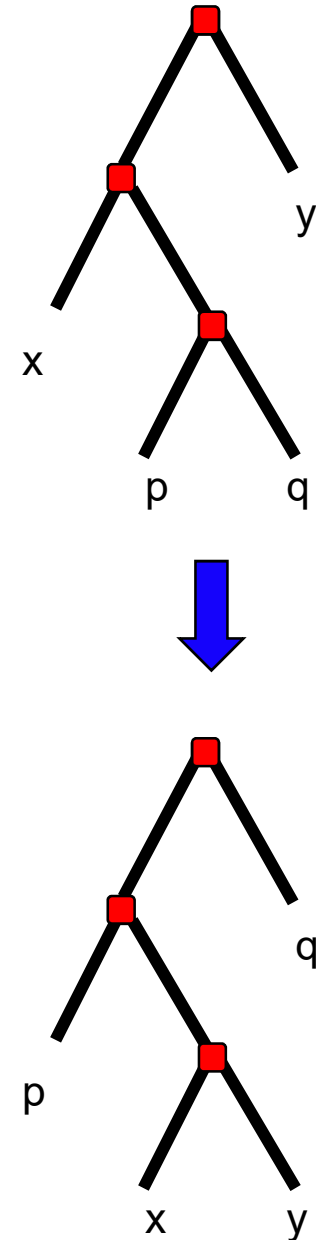
- **Claim 1** In an optimal tree code, every leaf has a sibling.
- **Proof** Otherwise, replace the lone leaf by its parent to get a tree code with lower cost.



$$cost(T) = 4 \cdot 2 + 3 \cdot 2 + 3 \cdot 2 + 3 \cdot 1 + 3 \cdot 1 = 26$$

# Huffman code is optimal

- **Claim 2** Consider the two least frequent letters  $x$  and  $y$ . In the an optimal tree code  $T$ ,  $x$  and  $y$  are **siblings** of each other at the **max depth** of the tree.
- **Proof** Suppose not, and let  $p$  be a node at the max depth.
  - $p$  has a sibling  $q$  by Claim 1.
  - $p$  and  $q$  have higher frequency than  $x$  and  $y$ , resp.
  - Create a new tree where we swap  $p$  and  $x$ , and  $q$  and  $y$ .
  - The new tree has strictly lower cost than  $T$ , since  $p, q$  have higher frequency than  $x, y$ . Contradiction.







# Huffman code is optimal

- **Thm** Huffman code is optimal.
- **Proof** Use induction on number of letters in the code. Suppose it's true up to  $n - 1$ .
  - Consider a code with  $n$  letters. Let  $x, y$  be the letters with the lowest frequency.
  - Let  $T$  be the Huffman code tree on the  $n$  letters.
    - Create a new node  $z$  with frequency  $f(z) = f(x) + f(y)$ .
    - Let  $S$  be a tree formed from  $T$  by removing  $x$  and  $y$ , and replacing their parent by  $z$ .
  - $S$  is the Huffman code on the  $n - 1$  letters.
    - Because of the recursive way Huffman encoding works.
  - $S$  is an optimal tree code on the  $n - 1$  letters, by induction.
  - $cost(T) = cost(S) + f(x) + f(y)$ .
    - All the nodes in  $S$  and  $T$  are the same, except  $x, y, z$ .
    - $cost(z) = d(z) \cdot f(z) = d(z) \cdot (f(x) + f(y))$ .
    - $d(z) = d(x) - 1 = d(y) - 1$ .
    - $cost(x) + cost(y) = cost(z) + f(x) + f(y)$ .

# Huffman code is optimal

## ■ Proof (continued)

- Let  $T'$  be an optimal tree code on the  $n$  letters.
  - By Claim 2,  $x$  and  $y$  are siblings in  $T'$ .
  - Merge them into a node  $z'$ , with  $f(z') = f(x) + f(y)$ . Form a tree  $S'$  by removing  $x$  and  $y$  from  $T'$ , and replacing their parent by  $z'$ .
  - $S'$  is a tree code on  $n - 1$  letters.
- $cost(T') = cost(S') + f(x) + f(y) \geq cost(S) + f(x) + f(y) = cost(T)$ .
  - First equality because  $x, y$  at depth one greater than  $z$ .
  - First inequality because  $S$  is opt tree code on  $n - 1$  letters.
- So, the tree  $T$  produced by Huffman encoding is optimal.