



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

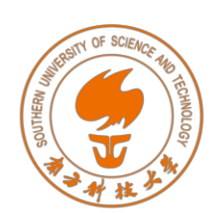
# Algorithm Design and Analysis (H)

CS216

Instructor: Shan CHEN (陈杉)

[chens3@sustech.edu.cn](mailto:chens3@sustech.edu.cn)

(slides edited from Prof. Shiqi Yu)



# Greedy Algorithms



# 8. Huffman Codes



# Encoding







- **Q.** Why do we need encoding?
- **A.** Encoding transforms data of human language to numbers such that they can be processed by digital computers.
- Ex. Postcode, character codes (Unicode), etc.
  
- **Q.** Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?
- **A.** We can encode  $2^5$  different symbols using a fixed length of 5 bits per symbol. This is called **fixed length encoding**.

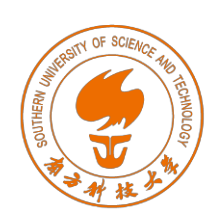




# Fixed Length Encoding

- 64 samples (1 poison)
- How many guinea pigs do we need to find the poison?

						
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
...						
63	1	1	1	1	1	1



# Data Compression

- **Q.** Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?
- **A.** Encode such characters with **fewer** bits and the others with more bits.
- **Q.** How do we know when the next symbol begins?
- **A.** Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that **no** code is a **prefix** of another one.
- **Ex.**  $c(a) = 01$ ,  $c(b) = 010$ ,  $c(e) = 1$ .
- **Q.** What is 0101?





# Prefix Codes

- **Def.** A **prefix code** for a set  $S$  is a function  $\gamma$  that maps each  $x \in S$  to a bit string such that for  $x, y \in S$ ,  $x \neq y$ ,  $\gamma(x)$  is **not a prefix** of  $\gamma(y)$ .
- **Ex.**  $c(a) = 11$ ,  $c(e) = 01$ ,  $c(k) = 001$ ,  $c(l) = 10$ ,  $c(u) = 000$ .
- Q. What is the meaning of 1001000001?
- A. "leuk"
- **Ex.** Frequencies in 1G text:  $f_a = 0.4$ ,  $f_e = 0.2$ ,  $f_k = 0.2$ ,  $f_l = 0.1$ ,  $f_u = 0.1$ .
- Q. What is the size of the encoded text?
- A.  $2f_a + 2f_e + 3f_k + 2f_l + 3f_u = 2.3G$





# Optimal Prefix Codes

- **Def.** The **average bits per letter (ABL)** of a prefix code  $\gamma$  is the sum, over all symbols  $x \in S$ , of its frequency  $f_x$  times the number of bits of its encoding  $\gamma(x)$ :

$$ABL(\gamma) = \sum_{x \in S} f_x |\gamma(x)|$$

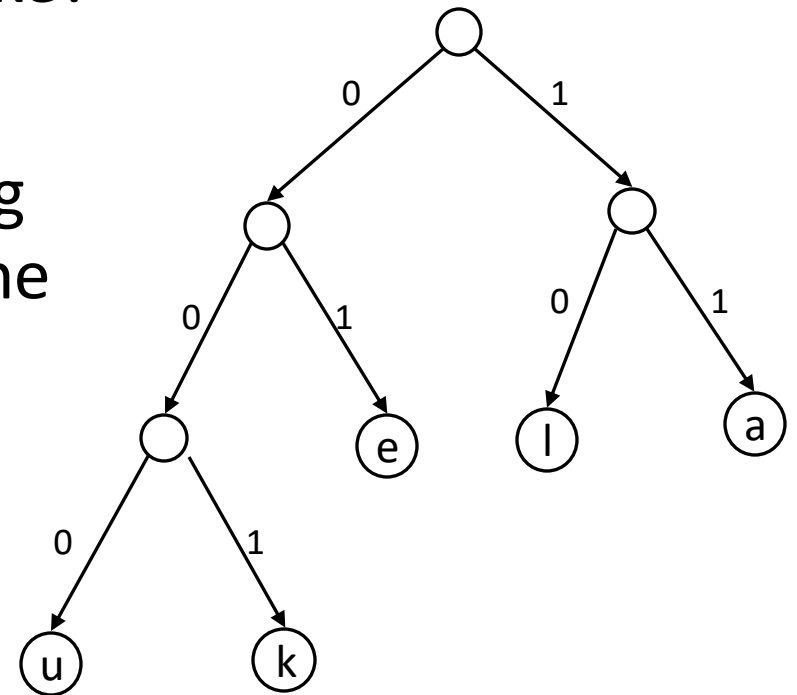
- **Q.** Can we construct a prefix code that has the lowest ABL?
- **Observation.** A prefix code can be modeled in a binary tree.
  - Any binary code can be modeled in a binary tree.





# Representing Prefix Codes with Binary Trees

- **Ex.**  $c(a) = 11$ ,  $c(e) = 01$ ,  $c(k) = 001$ ,  $c(l) = 10$ ,  $c(u) = 000$ .
- **Q.** How does the tree of a prefix code look like?
- **A.** Only the **leaves** have a label.
- **Pf.** An encoding of  $x$  is a prefix of an encoding of  $y$  if and only if the path of  $x$  is a prefix of the path of  $y$ .

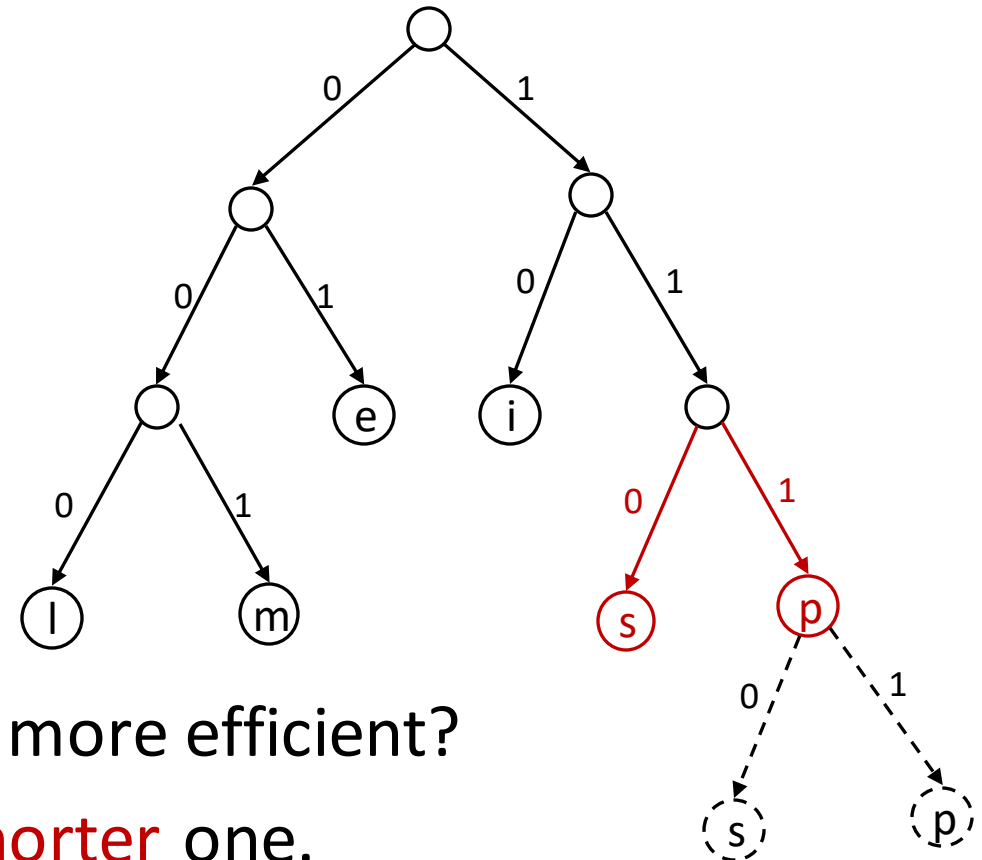






# Representing Prefix Codes with Binary Trees

- **Q.** What is the meaning of 111010001111101000?
- **A.** "simplel"



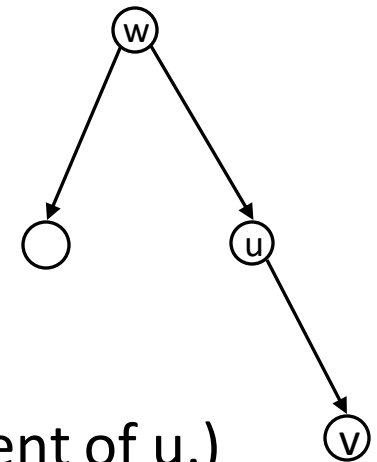
- **Q.** How can this prefix code be made more efficient?
- **A.** Change encoding of s and p to a **shorter** one.





# Representing Prefix Codes with Binary Trees

- **Def.** A tree is **full** if every node that is not a leaf has two children.
- **Claim.** The binary tree corresponding to an **optimal** prefix code is full.
- **Pf. (by contradiction)**
  - Suppose  $T$  is binary tree of optimal prefix code and is not full.
  - This means there is a node  $u$  with only one child  $v$ .
  - Case 1:  $u$  is the root.
    - ✓ Delete  $u$  and use  $v$  as the root.
  - Case 2:  $u$  is not the root.
    - ✓ Delete  $u$  and make  $v$  be a child of  $w$  in place of  $u$ . ( $w$  is the parent of  $u$ .)
  - In both cases the number of bits needed to encode any leaf in the subtree of  $v$  is **decreased**. The rest of the tree is not affected.
  - Clearly this new tree  $T'$  has a smaller ABL than  $T$ . Contradiction!

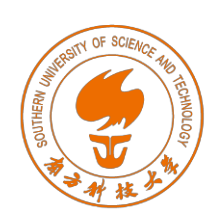




# Optimal Prefix Codes: False Start

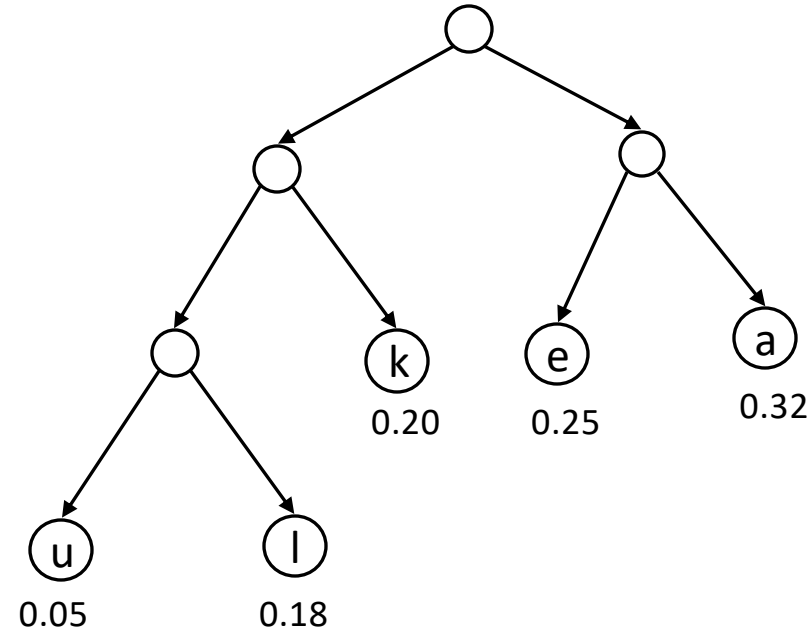
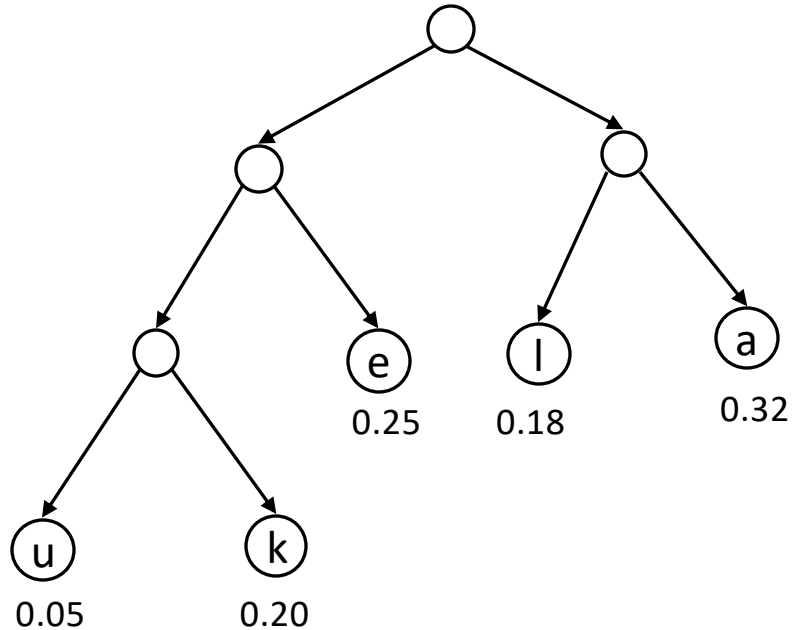
- **Q.** Where in the tree of an optimal prefix code should letters with a high frequency be placed?
- **A.** Near the top.
- **Greedy template.** [Shannon-Fano, 1949] Create tree **top-down**. Split  $S$  into two sets  $S_1$  and  $S_2$  with (almost) equal frequencies. Recursively build tree for  $S_1$  and  $S_2$ .
- **Q.** Does this approach output an optimal prefix code?





# Optimal Prefix Codes: False Start

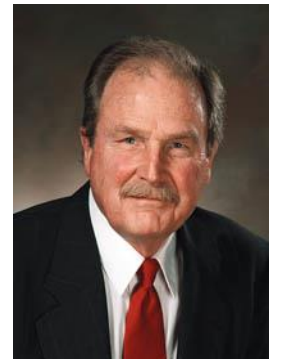
- **Greedy template.** [Shannon-Fano, 1949] Create tree **top-down**. Split  $S$  into two sets  $S_1$  and  $S_2$  with (almost) equal frequencies. Recursively build tree for  $S_1$  and  $S_2$ .
- **Counterexample:**  $f_a = 0.32$ ,  $f_e = 0.25$ ,  $f_k = 0.20$ ,  $f_l = 0.18$ ,  $f_u = 0.05$





# Optimal Prefix Codes: Huffman Encoding

- **Observation.** Lowest frequency items should be at the lowest level in the tree of an optimal prefix code.
- **Observation.** For  $n > 1$ , the lowest level always contains at least two leaves.
- **Observation.** The order in which items appear in a level does not matter.
- **Claim.** There is an optimal prefix code with tree  $T^*$  where the **two lowest-frequency letters** are assigned to **leaves that are siblings** in  $T^*$ .
- **Greedy template.** [Huffman, 1952] Create tree **bottom-up**. Make two leaves for two lowest-frequency letters  $y$  and  $z$ . Recursively build tree for the rest using a meta-letter for  $y, z$ .





# Optimal Prefix Codes: Huffman Encoding

```
Huffman(S) {  
  if |S| == 2 {  
    return tree with root and 2 leaves  
  } else {  
    let y and z be lowest-frequency letters in S  
    let S' be S with y and z removed  
    insert new letter  $\omega$  in S' with  $f_{\omega} = f_y + f_z$   
    T' = Huffman(S')  
    T = add two children y and z to leaf  $\omega$  from T'  
    return T  
  }  
}
```

- **Q.** What is the time complexity?
- **A.**  $T(n) = T(n - 1) + O(\log n)$ , therefore  $T(n) = O(n \log n)$ .

↖ ExtractMin from priority queue S







# Huffman Encoding: Greedy Analysis

- **Claim.**  $ABL(T) = ABL(T') + f_\omega$ .
- Pf. Recall that  $T = T'$  with two children  $x, y$  added to leaf  $\omega$ .

$$\begin{aligned} ABL(T) &= \sum_{x \in S} f_x \cdot \text{depth}_T(x) \\ &= f_{y^*} \cdot \text{depth}_T(y^*) + f_{z^*} \cdot \text{depth}_T(z^*) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_T(x) \\ &= (f_{y^*} + f_{z^*}) \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\ &= f_\omega \cdot (1 + \text{depth}_{T'}(\omega)) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\ &= f_\omega + f_\omega \cdot \text{depth}_{T'}(\omega) + \sum_{x \neq y^*, z^*} f_x \cdot \text{depth}_{T'}(x) \\ &= f_\omega + \sum_{x \in S'} f_x \cdot \text{depth}_{T'}(x) \\ &= f_\omega + ABL(T'). \quad \blacksquare \end{aligned}$$





# Huffman Encoding: Greedy Analysis

- **Claim.** Huffman code for  $S$  achieves the **minimum ABL** of any **prefix code**.
- Pf. (by induction over  $n = |S|$ )
  - **Basis Step:** For  $n = 2$  there is no shorter code than root and two leaves.
  - **Inductive Hypothesis (IH):** Suppose Huffman tree  $T'$  for  $S'$  of size  $n - 1$ , with leaf  $\omega$  instead of the lowest-frequency nodes  $y$  and  $z$ , is optimal.
  - **Inductive Step:** (by contradiction) [proof idea]
    - ✓ Suppose some other tree  $Z$  of size  $n$  is better.
    - ✓ Delete lowest frequency items  $y$  and  $z$  from  $Z$  creating  $Z'$ .
    - ✓  $Z'$  cannot be better than  $T'$  by IH.





# Huffman Encoding: Greedy Analysis

- **Claim.** Huffman code for  $S$  achieves the **minimum ABL** of any **prefix code**.
- Pf. (by induction over  $n = |S|$ )
  - **Basis Step:** For  $n = 2$  there is no shorter code than root and two leaves.
  - **Inductive Hypothesis (IH):** Suppose Huffman tree  $T'$  for  $S'$  of size  $n - 1$ , with leaf  $\omega$  instead of the lowest-frequency nodes  $y$  and  $z$ , is optimal.
  - **Inductive Step: (by contradiction)**
    - ✓ Let  $T$  be the Huffman tree. Suppose there is tree  $Z$  such that  $ABL(Z) < ABL(T)$ .
    - ✓ Then w.l.o.g. we assume  $Z$  has leaves  $y$  and  $z$  as siblings (see Observation).
    - ✓ Let  $Z'$  be  $Z$  with  $y$  and  $z$  deleted and their former parent labeled  $\omega'$ .
    - ✓ We know that  $ABL(Z) = ABL(Z') + f_{\omega'}$ , as well as  $ABL(T) = ABL(T') + f_{\omega}$ .
    - ✓ Recall  $ABL(Z) < ABL(T)$  and  $f_{\omega'} = f_{\omega}$ , so  $ABL(Z') < ABL(T')$ . Contradiction with IH!



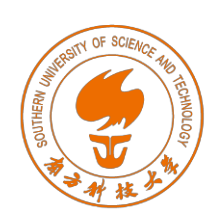


# Huffman Encoding: Summary

- **Greedy approach:** Shrink the size of the problem instance, so that an equivalent smaller problem can then be solved by recursion.
  - The greedy operation is proved to be "safe": solving the smaller problem still leads to an optimal solution for the original problem.
- **Application.** ZIP file format that supports lossless data compression.
  - Its most common compression algorithm DEFLATE uses LZ77 and Huffman.
  - Document: <https://pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.2.0.txt>

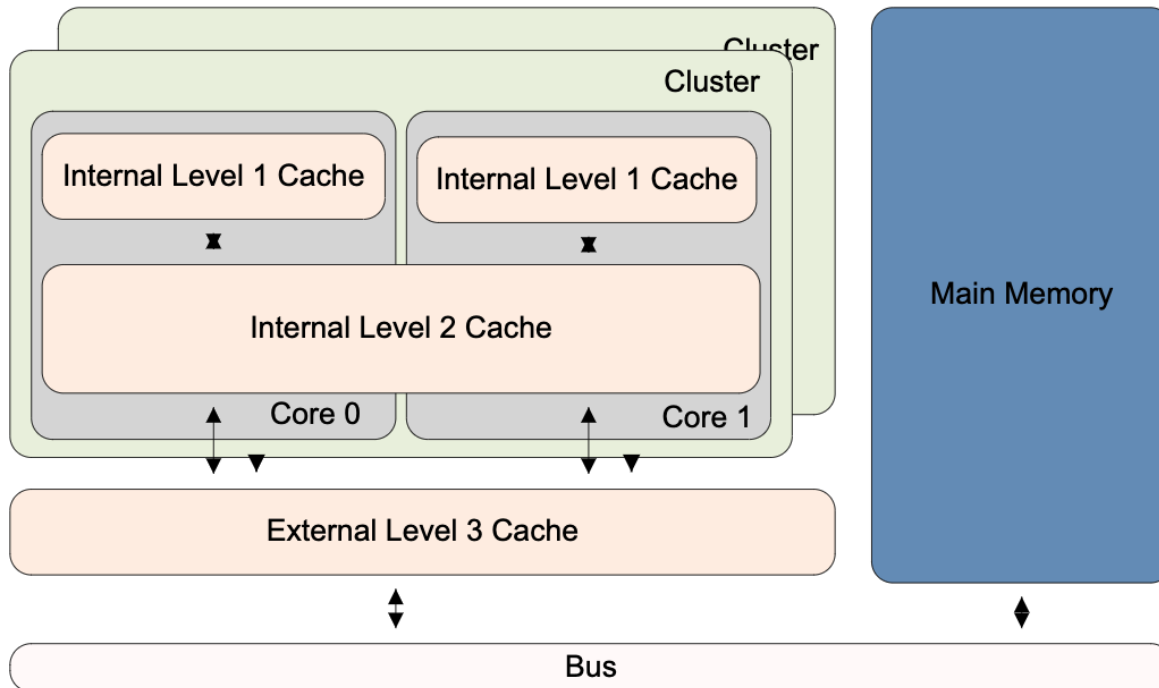


# 9. Optimal Caching



# Caching

- HD->memory->cache



Memory type	Typical size	Typical access time
Processor registers	128KB	1 cycle
On-chip L1 cache	32KB	1-2 cycle(s)
On-chip L2 cache	128KB	8 cycles
Main memory (L3) dynamic RAM	MB or GB <sup>[1]</sup>	30-42 cycles
Back-up memory (hard disk) (L4)	MB or GB	> 500 cycles

<sup>[1]</sup> Size limited by the processor core addressing, for example a 32-bit processor without memory management can directly address 4GB of memory.



# Optimal Offline Caching

- **Offline caching.**

- Cache with capacity to store  $k$  items.
- Sequence of  $m$  item **requests**  $d_1, d_2, \dots, d_m$ . ← offline: known sequence
- Cache **hit**: item already in cache when requested.
- Cache **miss**: item not already in cache when requested.
  - ✓ Must bring requested item into cache, and evict some existing item if full.

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

- **Ex:**  $k = 2$ , initial cache =  $ab$ , requests:  $a, b, c, b, c, a, a, b$ .
- Optimal eviction schedule: 2 cache misses.



# Optimal Offline Caching

- **Greedy templates.**

- LIFO. (last-in-first-out) Evict item brought in least recently.
- FIFO. (first-in-first-out) Evict item brought in most recently.
- LRU. (least-recently-used) Evict item whose most recent access was earliest.
- LFU. (least-frequently-used) Evict item that was least frequently requested.





# Optimal Offline Caching

- Greedy templates.

- LIFO. (last-in-first-out)
- FIFO. (first-in-first-out)
- LRU. (least-recently-used)
- LFU. (least-frequently-used)

		cache							
requests	⋮	.	.	.	.	.			
	<i>a</i>	<i>a</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>		FIFO: eject <i>a</i>	
	<i>d</i>	<i>a</i>	<i>w</i>	<i>x</i>	<i>d</i>	<i>z</i>		LRU: eject <i>d</i>	
	<i>a</i>	<i>a</i>	<i>w</i>	<i>x</i>	<i>d</i>	<i>z</i>			
	<i>b</i>	<i>a</i>	<i>b</i>	<i>x</i>	<i>d</i>	<i>z</i>			
	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>z</i>			
	<i>e</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>		LIFO: eject <i>e</i>	
	<i>g</i>	?	?	?	?	?			
	<i>b</i>								cache miss (which item to eject?)
	<i>e</i>								
	<i>d</i>								
	⋮								



- 清除缓存中直到最远将来被请求的项

future queries: **g** a b c e d a b b a c d e a **f** a d e f g h ...

↑  
cache miss

↑  
eject this one

- **Theorem.** [Bellady, 1960s] FF is optimal eviction schedule.
- Algorithm and theorem are intuitive; proof is subtle (shown later).



# Reduced Eviction Schedules

- **Def.** A **reduced** schedule is a schedule that only inserts an item into the cache in a step when that item is **requested** and **not yet in cache**.

<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>c</i>	<i>a</i>	<i>d</i>	<i>c</i>
<i>d</i>	<i>a</i>	<i>d</i>	<i>c</i>
<i>a</i>	<i>a</i>	<i>c</i>	<i>b</i>
<i>b</i>	<i>a</i>	<i>c</i>	<i>b</i>
<i>c</i>	<i>a</i>	<i>c</i>	<i>b</i>
<i>d</i>	<i>d</i>	<i>c</i>	<i>b</i>
<i>d</i>	<i>d</i>	<i>c</i>	<i>d</i>

an unreduced schedule

*d* enters cache  
without a request  
*unnecessary step*

*d* enters cache  
even though already  
in cache

<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>d</i>	<i>a</i>	<i>d</i>	<i>c</i>
<i>a</i>	<i>a</i>	<i>d</i>	<i>c</i>
<i>b</i>	<i>a</i>	<i>d</i>	<i>b</i>
<i>c</i>	<i>a</i>	<i>c</i>	<i>b</i>
<i>d</i>	<i>d</i>	<i>c</i>	<i>b</i>
<i>d</i>	<i>d</i>	<i>c</i>	<i>b</i>

a reduced schedule

*perform better*



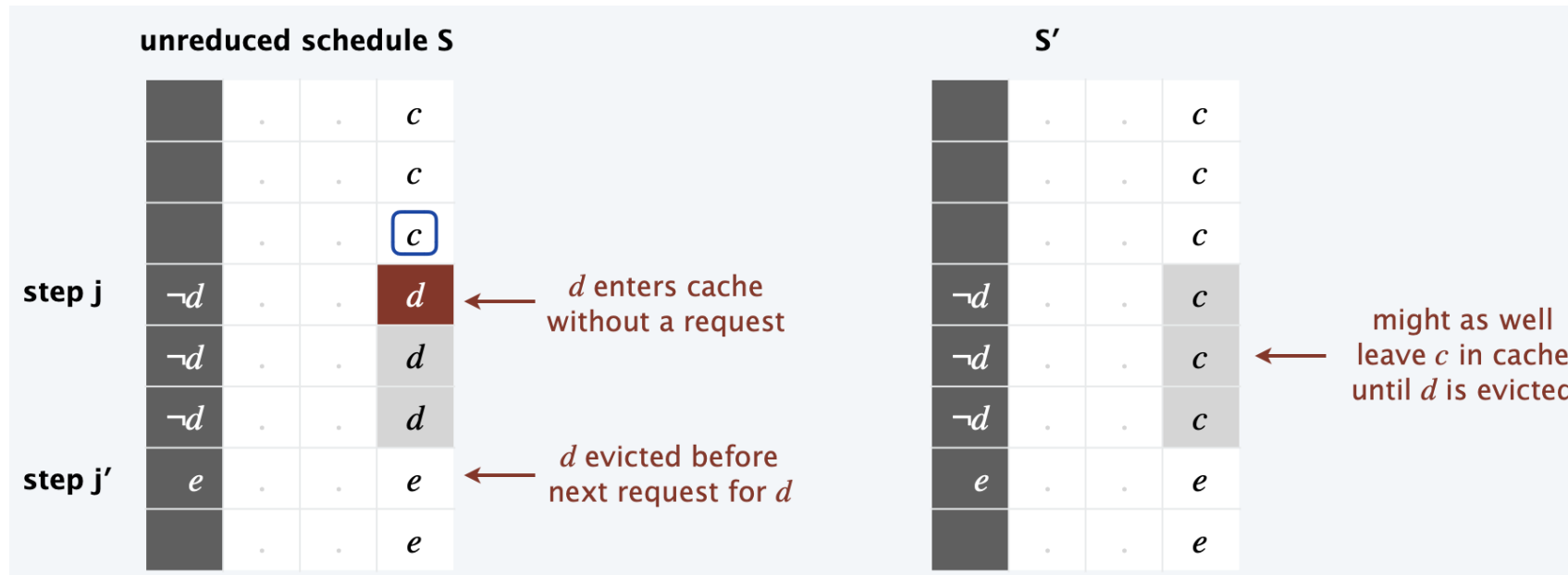
# Reduced Eviction Schedules

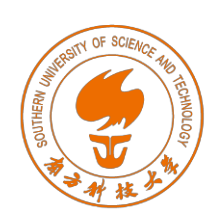
- **Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more evictions.
- Pf. (by induction on number of steps  $j$ ) [proof idea]
  - Base case:  $j = 0$ .
  - Case 1:  $S$  brings  $d$  into the cache in step  $j$  without a request.
  - Case 2:  $S$  brings  $d$  into the cache in step  $j$  even though  $d$  is in cache.
  - If multiple unreduced items in step  $j$ , apply each one in turn.
  - Deal with Case 1 before Case 2, as resolving Case 1 might trigger Case 2 ■



# Reduced Eviction Schedules

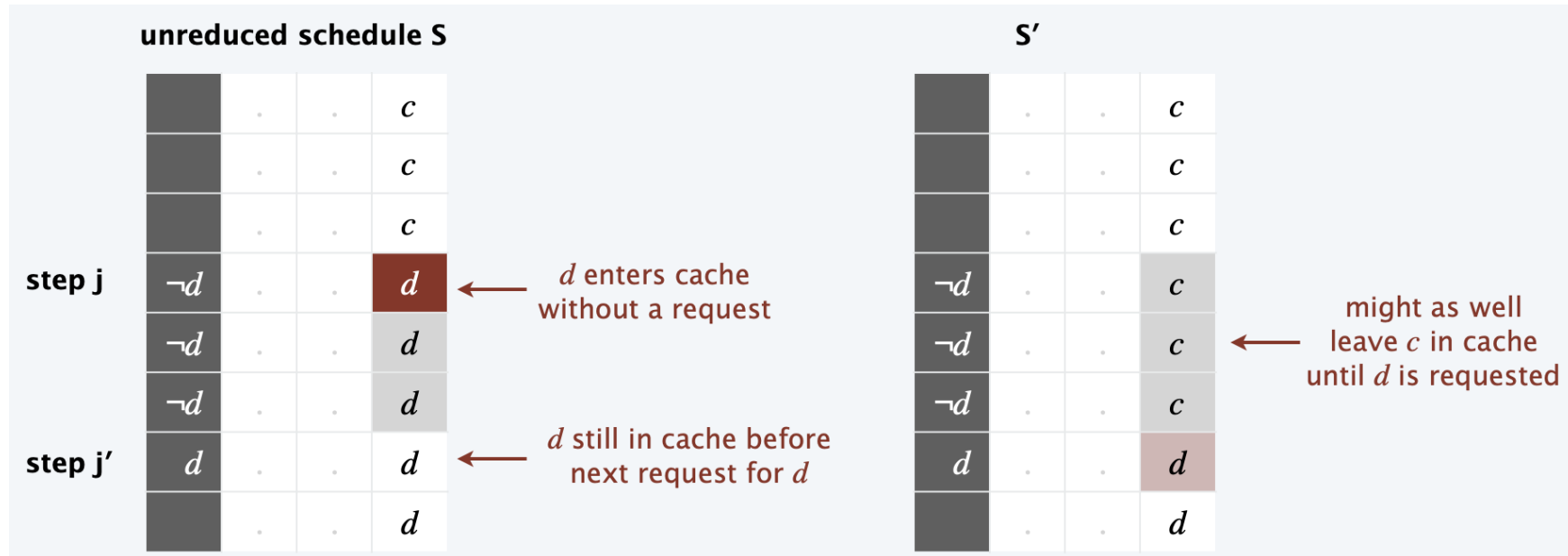
- **Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more evictions.
- Pf. (by induction on number of steps  $j$ ) [ $d$  is inserted in step  $j$ ]
  - Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache.
  - Case 1a:  $d$  evicted before next request for  $d$ .





# 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 steps  $j$ ) [ $d$  is inserted in step  $j$ ]
  - Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache.
  - Case 1b: next request for  $d$  occurs before  $d$  is evicted.





# 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 steps  $j$ ) [ $d$  is inserted in step  $j$ ]
  - Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache.
  - Case 2a:  $d$  evicted before it is needed.

unreduced schedule $S$					$S'$				
step $j$		$d_1$	$a$	$c$			$d_1$	$a$	$c$
		$d_1$	$a$	$c$			$d_1$	$a$	$c$
		$d_1$	$a$	$c$			$d_1$	$a$	$c$
	$d$	$d_1$	$a$	$d_3$	←	$d$	$d_1$	$a$	$c$
	$d$	$d_1$	$a$	$d_3$	←	$d$	$d_1$	$a$	$c$
step $j'$	$c$	$c$	$a$	$d_3$		$c$	$c$	$a$	$c$
	$b$	$c$	$a$	$b$	←	$b$	$c$	$a$	$b$
	$d$	$c$	$a$	$d_3$	←	$d$	$c$	$a$	$d_3$

$d_3$  enters cache even though  $d_1$  is already in cache

$d_3$  not needed

$d_3$  evicted

$d_3$  needed

might as well leave  $c$  in cache until  $d_3$  is evicted



# 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 steps  $j$ ) [ $d$  is inserted in step  $j$ ]
  - Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache.
  - Case 2b:  $d$  needed before it is evicted.

unreduced schedule $S$					$S'$				
step $j$		$d_1$	$a$	$c$			$d_1$	$a$	$c$
		$d_1$	$a$	$c$			$d_1$	$a$	$c$
		$d_1$	$a$	$c$			$d_1$	$a$	$c$
	$d$	$d_1$	$a$	$d_3$		$d$	$d_1$	$a$	$c$
	$d$	$d_1$	$a$	$d_3$		$d$	$d_1$	$a$	$c$
	$c$	$c$	$a$	$d_3$		$c$	$c$	$a$	$c$
step $j'$	$a$	$c$	$a$	$d_3$		$a$	$c$	$a$	$c$
	$d$	$c$	$a$	$d_3$		$d$	$c$	$a$	$d_3$

$d_3$  enters cache even though  $d_1$  is already in cache

$d_3$  not needed

$d_3$  needed

might as well leave  $c$  in cache until  $d_3$  is needed





# Farthest-In-Future: Greedy Analysis

- **Theorem.** Farthest-in-future (FF) is optimal eviction algorithm.
- Pf. Follows directly from the following invariant.
- **Invariant.** There exists an optimal reduced schedule  $S$  that has the same eviction schedule as  $S_{FF}$  through the first  $j$  steps.
- Pf. (by induction on number of steps  $j$ )
  - **Basis Step:**  $j = 0$ .
  - **Inductive Step:** Let  $S$  be reduced schedule that satisfies invariant through  $j$  steps. We produce  $S'$  that satisfies invariant after  $j + 1$  steps.
    - ✓ Let  $d$  denote the item requested in step  $j + 1$ .
    - ✓ Since  $S$  and  $S_{FF}$  have agreed up until now, they have the same cache contents before step  $j + 1$ .



# Farthest-In-Future: Greedy Analysis

- **Theorem.** Farthest-in-future (FF) is optimal eviction algorithm.
- Pf. Follows directly from the following invariant.
- **Invariant.** There exists an optimal reduced schedule  $S$  that has the same eviction schedule as  $S_{FF}$  through the first  $j$  steps.
- Pf. (by induction on number of steps  $j$ ) [ $d$  is requested in step  $j + 1$ ]
  - **Inductive Step:** Let  $S$  be reduced schedule that satisfies invariant through  $j$  steps. We produce  $S'$  that satisfies invariant after  $j + 1$  steps.
  - Case 1:  $d$  is already in the cache.
    - ✓  $S' = S$  satisfies invariant.
  - Case 2:  $d$  is not in the cache and  $S$  and  $S_{FF}$  evicts the same item.
    - ✓  $S' = S$  satisfies invariant.



# Farthest-In-Future: Greedy 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  $f$ .

j	same	e	f	same	e	f
	S			S'		
j+1	same	e	d	same	d	f
	S			S'		

- ✓ Now  $S'$  agrees with  $S_{FF}$  on first  $j + 1$  requests; we show that having  $f$  in cache is no worse than having  $e$  in cache. must involve  $e$  or  $f$  or both ↘
- ✓ Let  $S'$  behave the same as  $S$  until  $S'$  is forced to take a different action, because (Case 3a/3b) either  $e$  or  $f$  is requested or because (Case 3c)  $S$  evicts  $e$ .



# Farthest-In-Future: Greedy Analysis

- Pf. (continued) [Let  $j'$  be the first step after  $j + 1$  that  $S$  and  $S'$  must take different actions, and let  $g$  be the item requested at step  $j'$ .]

- Case 3:  $d$  is not in the cache;  $S_{FF}$  evicts  $e$ ;  $S$  evicts  $f \neq e$ ;  $S'$  evicts  $e$ .

$j'$	same	$e$	same	$f$
------	------	-----	------	-----

$S$

$S'$

$S'$  agrees with  $S_{FF}$  through step  $j+1$

- Case 3a:  $g = e$ .
  - ✓ Can't happen with FF since there must be a request for  $f$  before  $e$ .
- Case 3b:  $g = f$ .
  - ✓ Element  $f$  is not 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 cache; now  $S$  and  $S'$  have the same cache.
  - ✓ Let  $S'$  behave exactly like  $S$  for remaining requests.

$S'$  is no longer reduced, but can be transformed into a reduced schedule that agrees with  $S_{FF}$  through step  $j+1$



# Farthest-In-Future: Greedy Analysis

- Pf. (continued) [Let  $j'$  be the **first** step after  $j + 1$  that  $S$  and  $S'$  must take different actions, and let  $g$  be the item requested at step  $j'$ .]
  - Case 3:  $d$  is not in the cache;  $S_{FF}$  evicts  $e$ ;  $S$  evicts  $f \neq e$ ;  $S'$  evicts  $e$ .

$j'$	same	$e$	same	$f$
------	------	-----	------	-----

$S$

$S'$

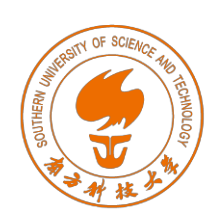
- Case 3c:  $S$  evicts  $e$  (and  $g \neq e, f$ ).
  - ✓ Make  $S'$  evict  $f$ .

$j'$	same	$g$	same	$g$
------	------	-----	------	-----

$S$

$S'$

- ✓ Now  $S$  and  $S'$  have the same cache.
- ✓ Let  $S'$  behave exactly like  $S$  for the remaining requests. ▀



# Optimal Caching: Summary

- **Online vs. offline caching.**

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

- **LRU.** Evict item 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.
- LIFO can be arbitrarily bad.
- LRU is  $k$ -competitive, i.e., for any sequence of requests  $R$ ,  $\text{LRU}(R) \leq k \text{FF}(R) + k$ .
- Randomized marking is  $O(\log k)$ -competitive.

*shown later in Section 13.8*