Computer Science 112
Data Structures

Lecture 16:
Hashing

Review: Huffman Encoding

Data Compression:

- **In most data some symbols appear more often than others**
  - **Eg English text 'e' appears more often than 'q'**
- **We can use this fact to represent data in fewer bits total**
  - **More frequent symbols: shorter codes**
  - **Less frequent symbols: longer codes**

Variable Length Takes Away Some Codes

- Suppose codes were

  1 = a, 11 = b
- Decode 111 as 'ab', as 'ba', or as 'aaa'?
- No character's code can be prefix of another

Variable Length Code

- Eg 4-symbols alphabet: {a, b, c, d} with frequencies:

  a: 50%, b: 30%, c: 10%, d:10%
- Variable length code: 1, 2, or 3 bits / character
  - e.g: 0 = a, 10 = b, 110 = c, 111 = d
  - aabcbaa = 00101101000
  - 11 bits / 7 characters = 1.6 bits/character
- Decode: 0100110 = 'abac'

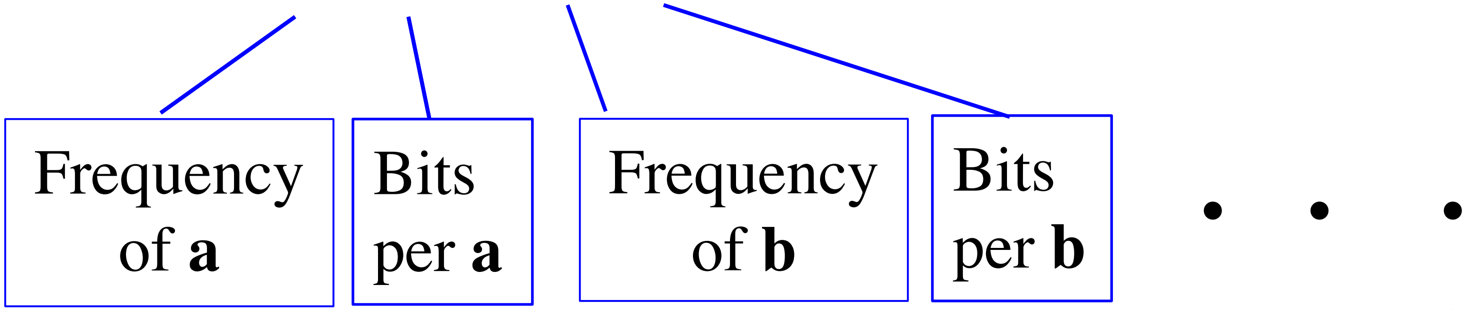
Variable Length Code

- **Eg 4-symbols alphabet: {a, b, c, d} with frequencies:**

  a: 50%, b: 30%, c: 10%, d:10%

- **Variable length code: 1, 2, or 3 bits / character**
  - **e.g: 0 = a, 10 = b, 110 = c, 111 = d**
- **Average bits per character:**

  0.5*1 + 0.3*2 + 0.1*3 + 0.1*3 = 1.7



Frequency of **a**

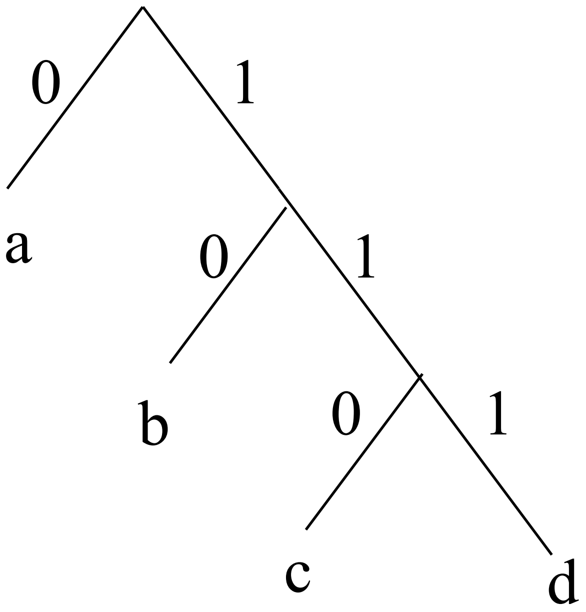
Bits per **a**

Frequency of **b**

Bits per **b**

. . .

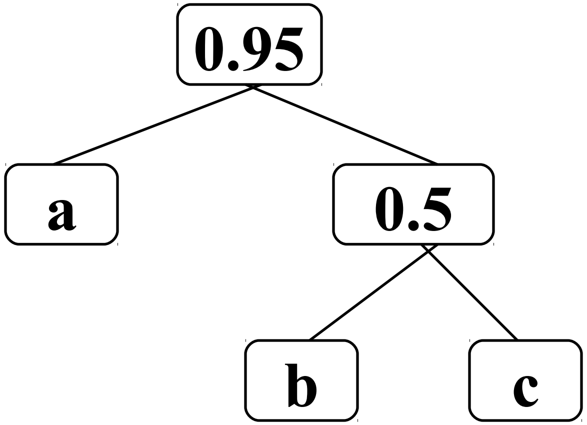
Huffman Code as a Tree



Symbols only at leaves

Algorithm to build tree

- 2 queues: S, T
- Contents of each queue: Tree
  - A leaf node stores (an index of) a symbol
  - A non-leaf mode stores total frequency of all symbols at leaves under this node
- E.g., for frequencies a: 0.45, b: 0.3, c: 0.2:



Algorithm to build tree

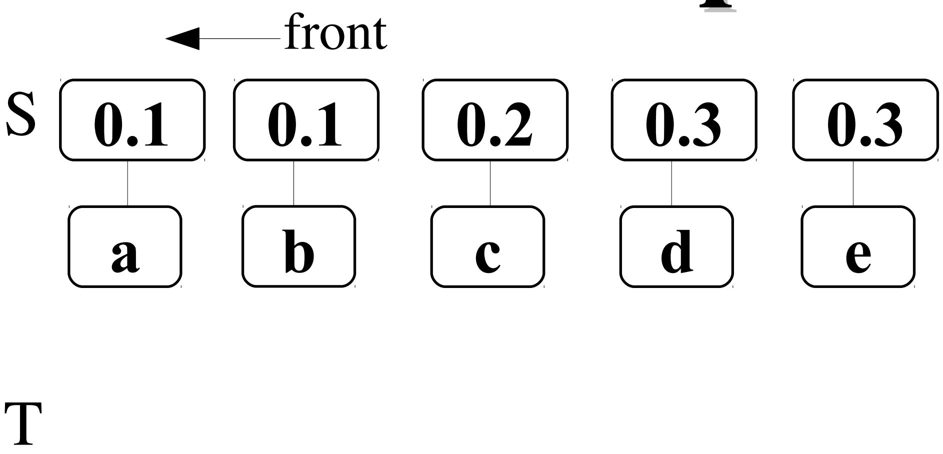
- 2 queues:
  - S initially holds 1-node trees for all symbols, least likely first
  - T empty

```
while not (S empty and T length == 1)
    find two least-weight trees in S, T and dequeue them
    make a tree with these two as subtrees
    enqueue this tree on T
read out codes from final tree
```

Example

a	.1
b	.1
c	.2
d	.3
e	.3

Example



front

S

0.1 a

0.1 b

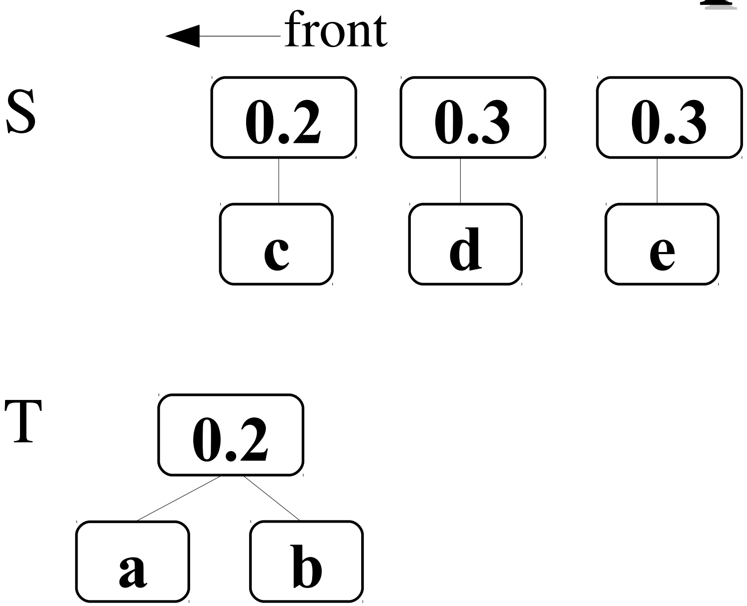
0.2 c

0.3 d

0.3 e

T

Example



front

S

0.2 0.3 0.3

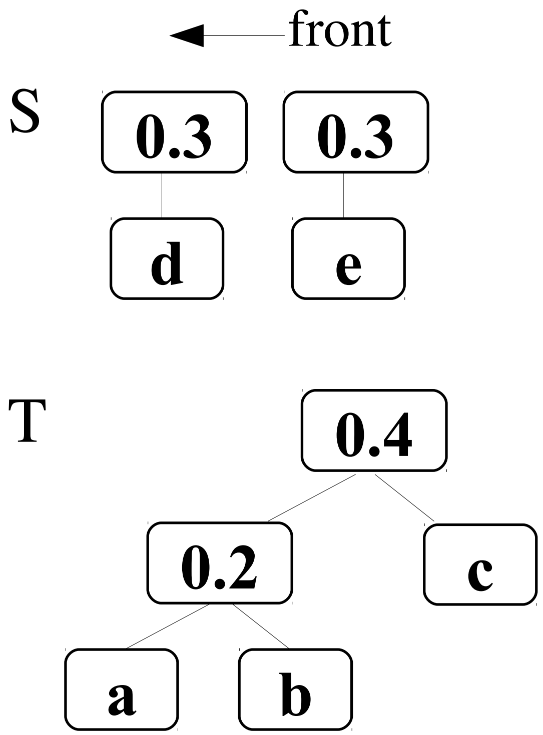
c d e

T

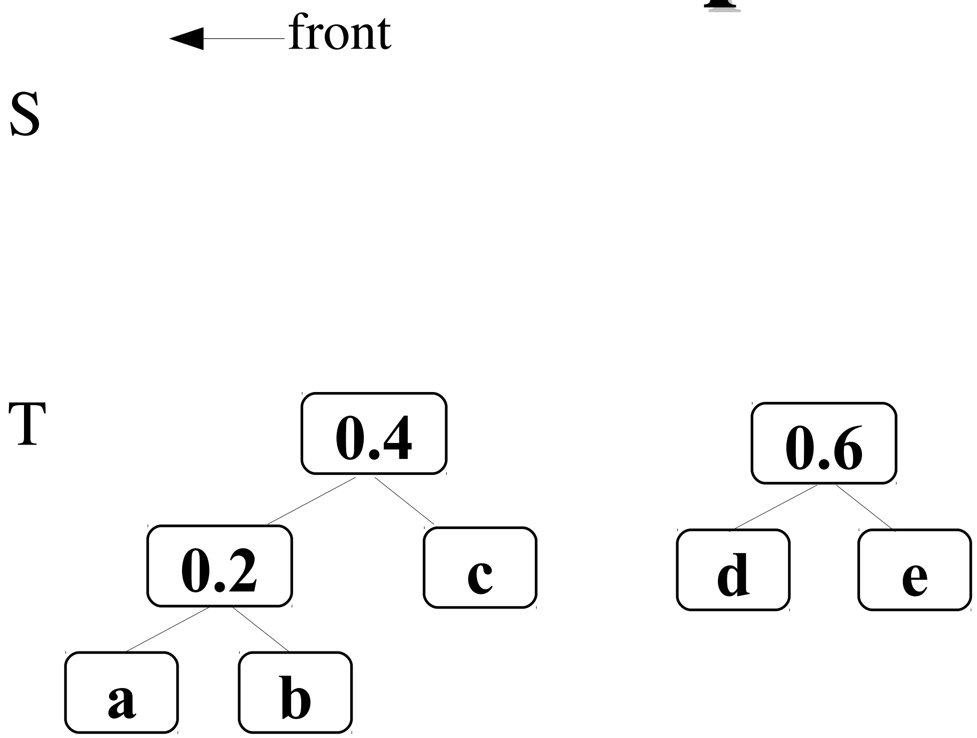
0.2

a b

Example



Example

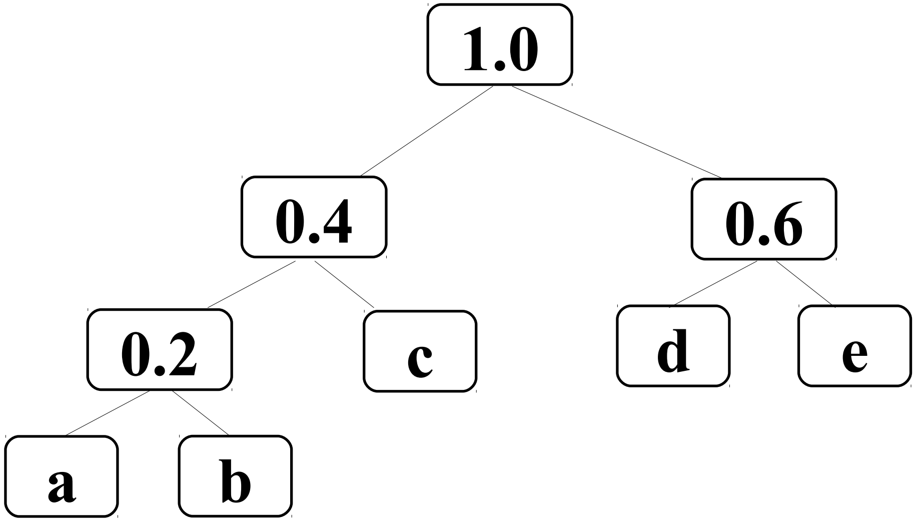


Example

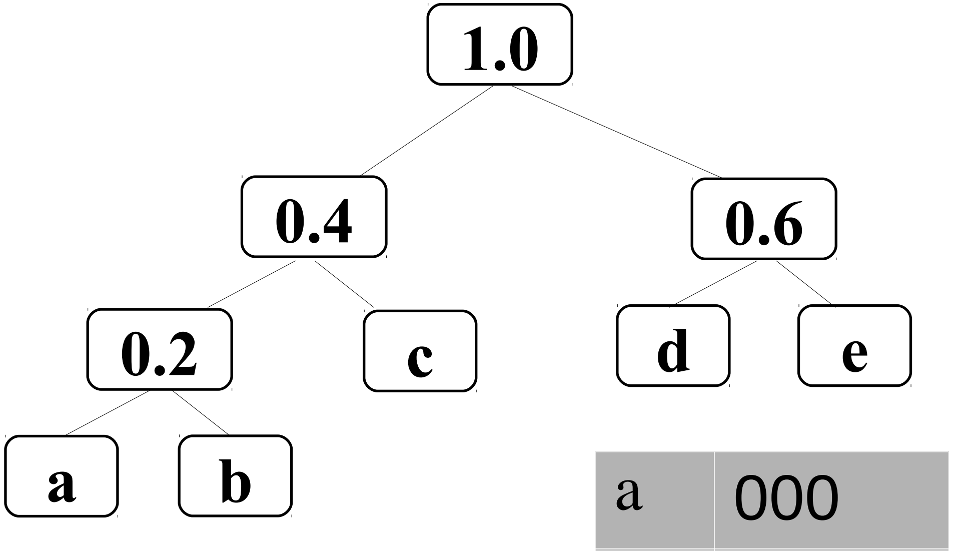
← front

S

T



Example



a	000
b	001
c	01
d	10
e	11

Big-O to build tree

- Create and enqueue one-symbol trees
  - O(n) where n is number of symbols in symbol set
- While >1 tree in queues

  dequeue 2 trees: 2*O(1) = O(1)

  merge into 1: create root, attach subtrees:

  enqueue 1 tree
  - O(n) iterations, O(1) work each → O(n) together
- grand total O(n)

Algorithm to encode

- For each symbol c in original string
  - find table entry for c
  - copy bits from table to encoded version of string
- let
  - n be size of symbol set
  - s be length of original version
  - b be length of encoded version
- O(s*n+b) or (s*log(n)+b)

Algorithm to decode

- Start at:
  - root of tree
  - beginning of encoded version
- For each bit in encoded version:
  - go left if bit is 0, right if 1

    when reach a leaf, output symbol at leaf and go back to root of tree
- O(b)

New: Hashing

- Suppose we want to store a set of numbers
  - add number to set, delete from set, test if in set should all be O(1)
- If range of numbers is small, e.g. 0 .. 9, we can use a boolean array, eg:

0	1	2	3	4	5	6	7	8	9
t	f	f	f	t	t	f	t	f	f

means {0, 4, 5, 7}

Hashing

- What if range of numbers is much larger than set size? eg
  - range 0 … 499,999
  - set size about 50
- If we use array of 500,000 elements, they will nearly all be false.

Hashing

- **Use an array of 500 objects**
  - **each array element <=> 1000 numbers**

index	corresponds to numbers	from	to
0		0	999
1		1,000	1,999
2		2,000	2,999
j		floor(j/1000)	floor(j/1000)+999
499		499,000	499,999

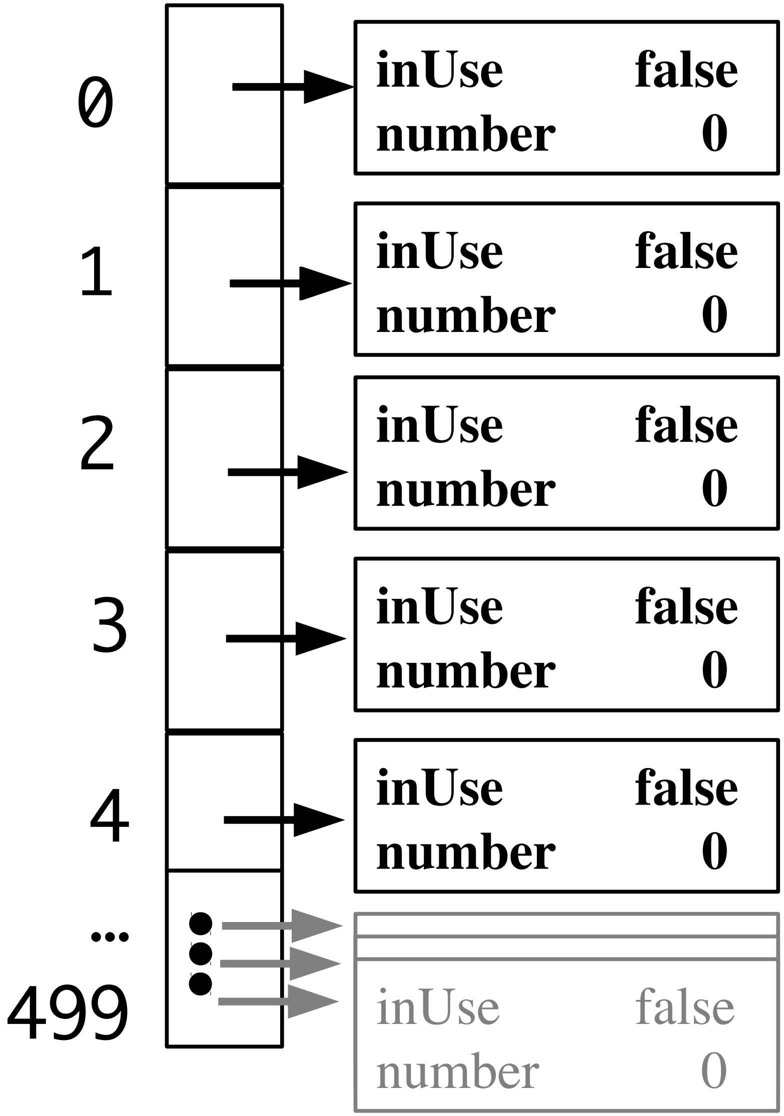
Hashing

- **The objects have instance variables:**
  - **boolean inUse**
  - **int number**

inUse	false
number	0

Hashing

- **The empty set**

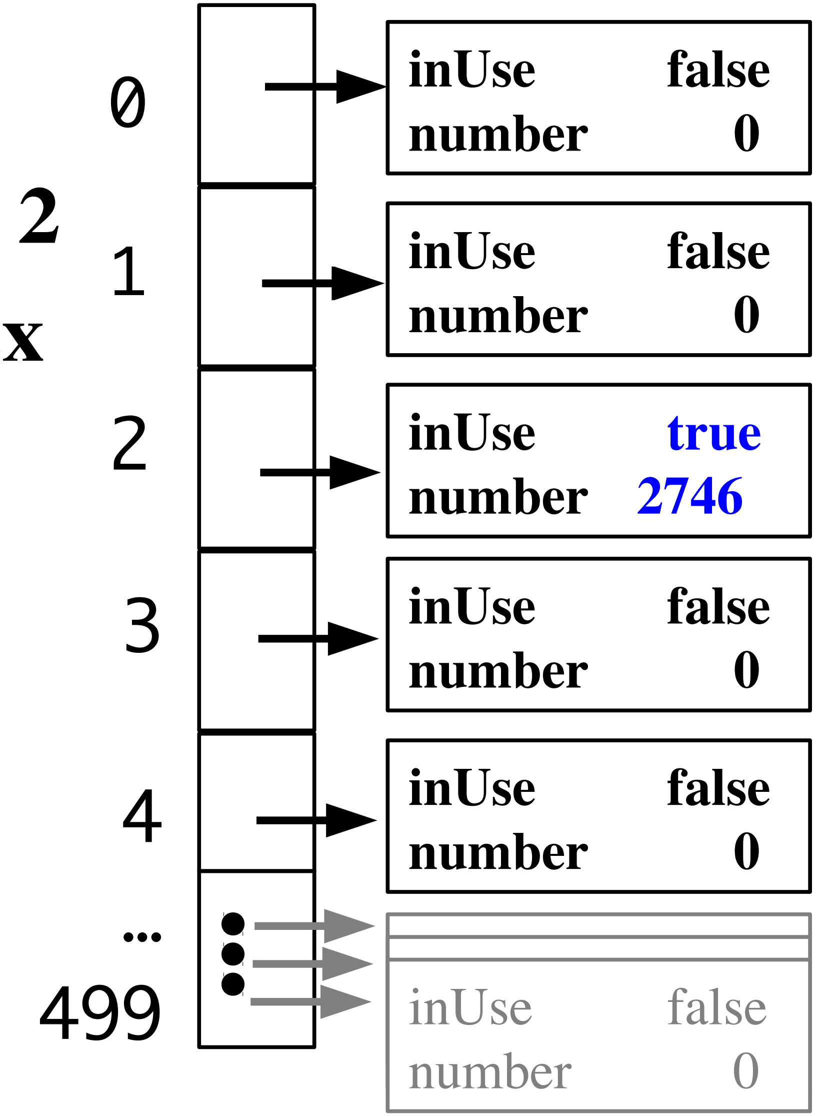


Hashing

Add 2746 to the set:

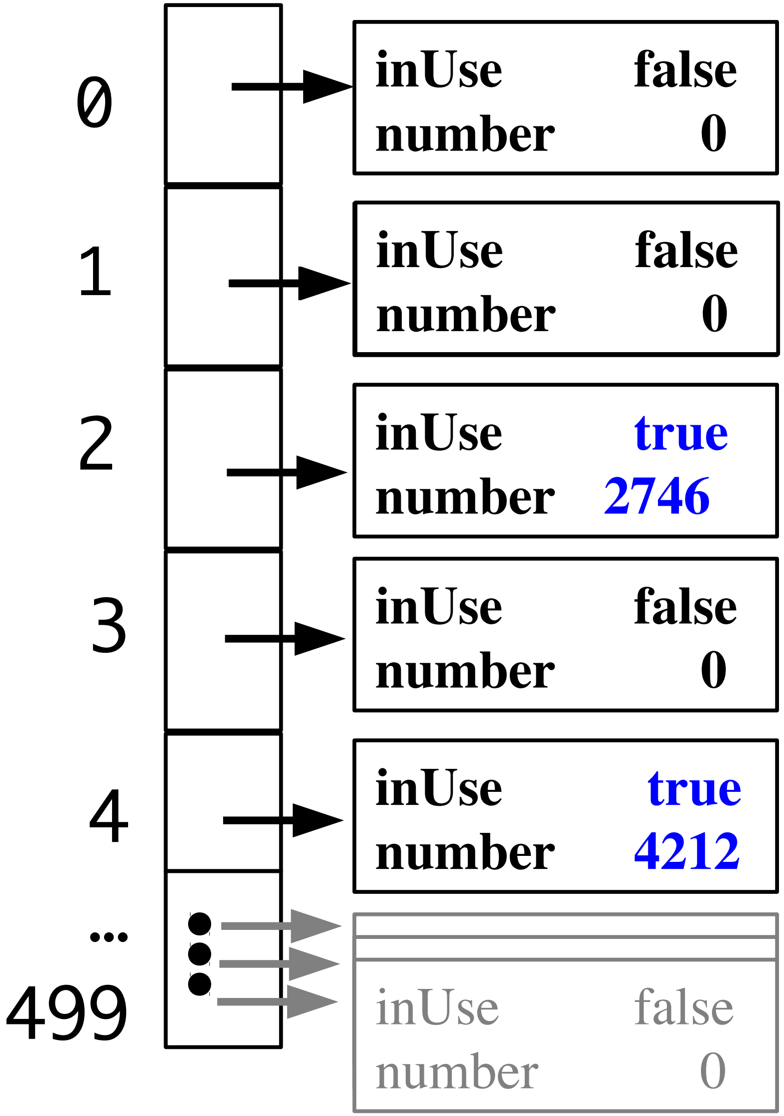
- **Corresponding index is 2**
- **Find object at that index**
- **Set inUse and number**

All O(1)



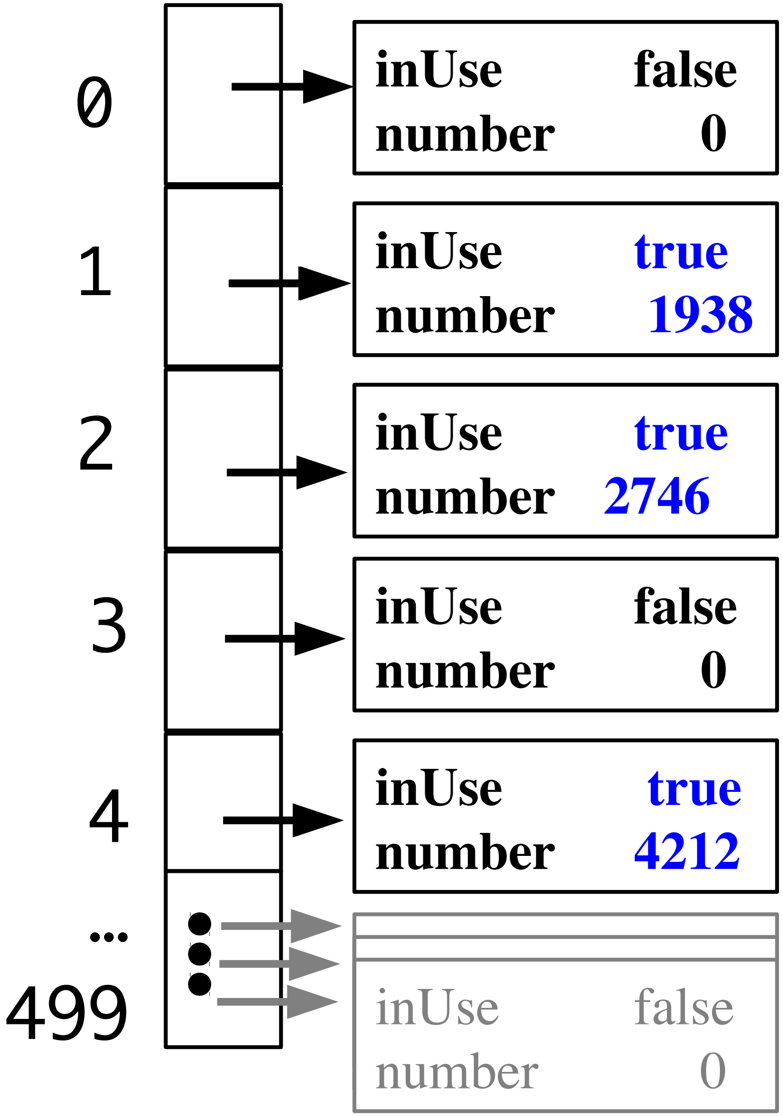
Hashing

Add 4212 to the set:



Hashing

Add 1938 to the set:

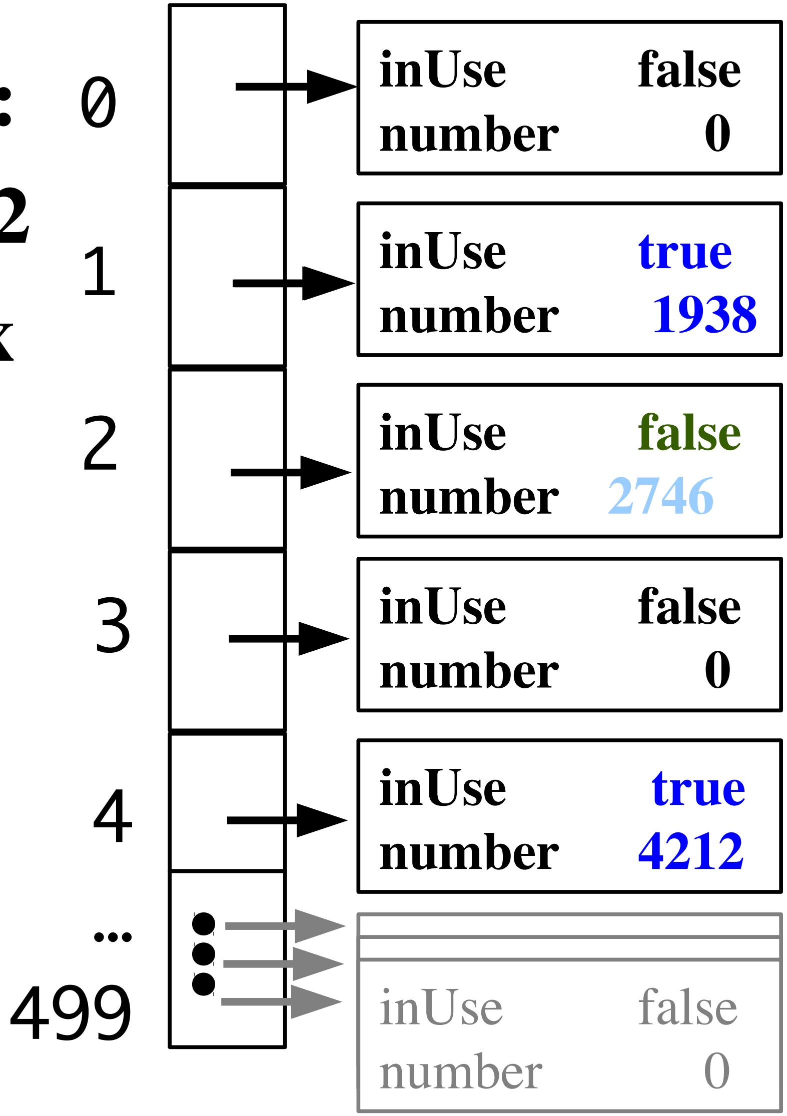


Hashing

Remove 2746 from the set:

- Corresponding index is 2
- Find object at that index
- Set inUse to false

All O(1)

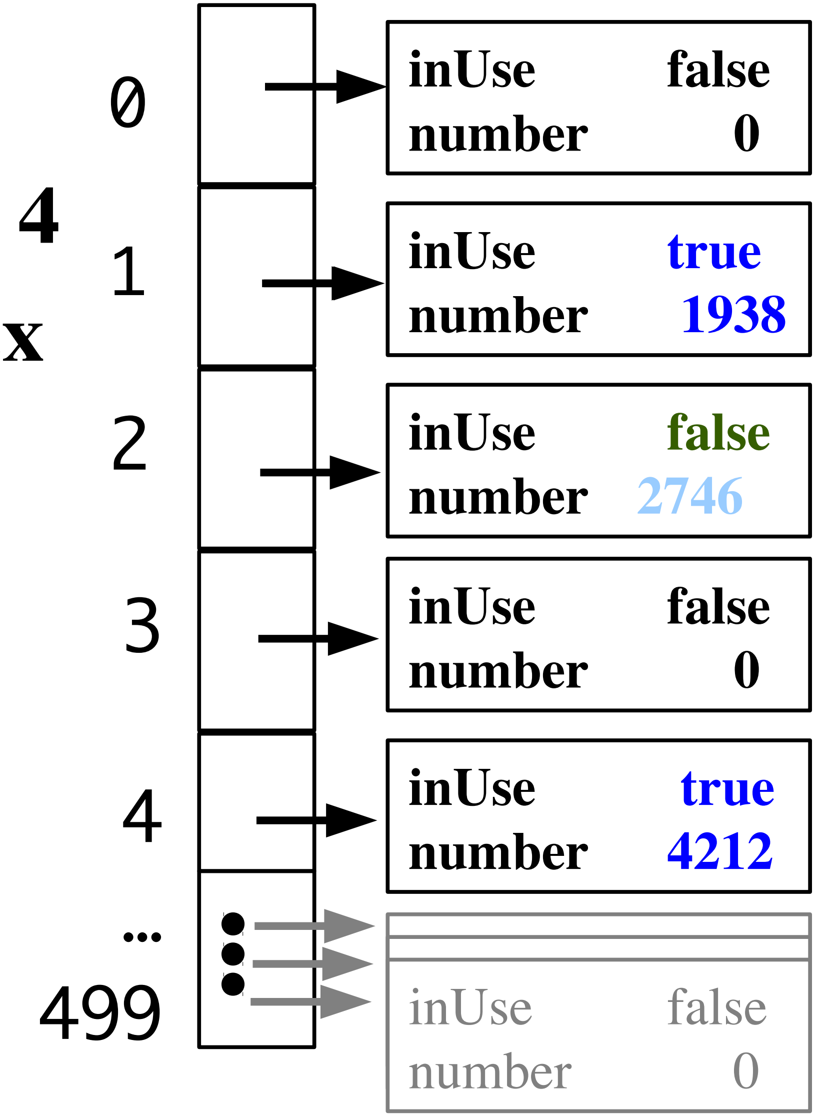


Hashing

Is 4352 in the set?

- Corresponding index is 4
- Find object at that index
- inUse is true but number ≠ 4352 → 4352 is not in the set

All O(1)

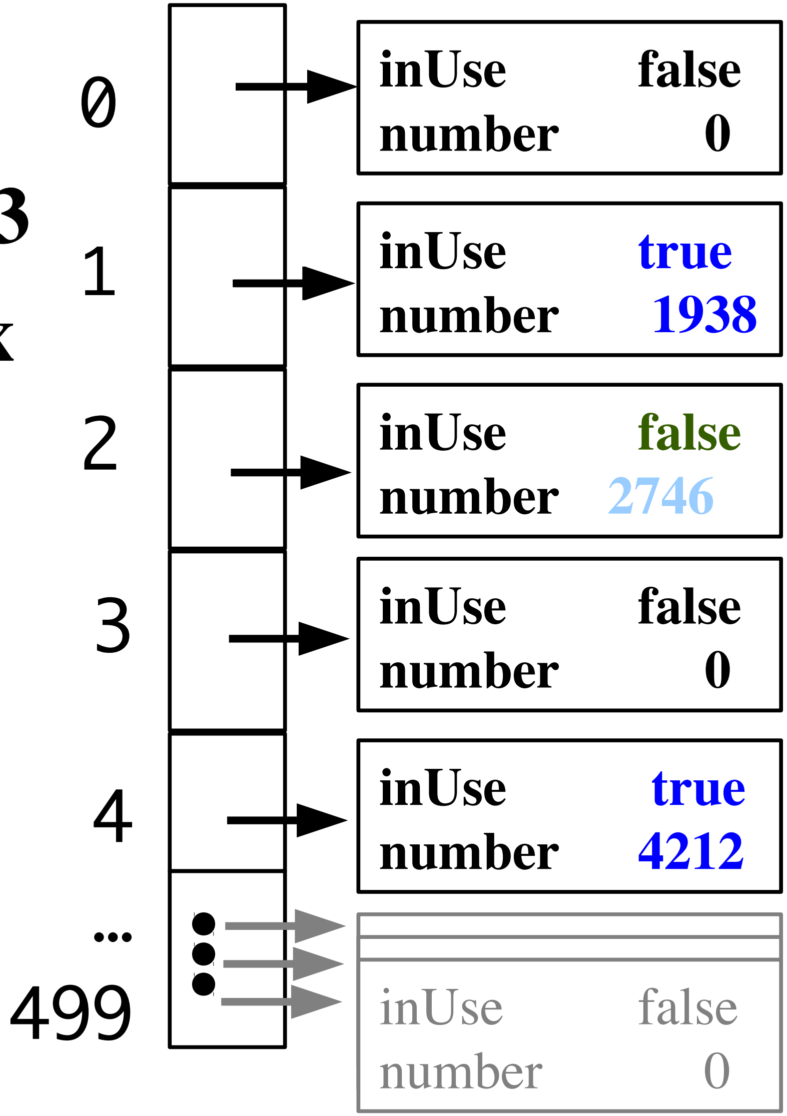


Hashing

Is 3314 in the set?

- Corresponding index is 3
- Find object at that index
- not inUse → 3314 is not in the set

All O(1)



Hash Function

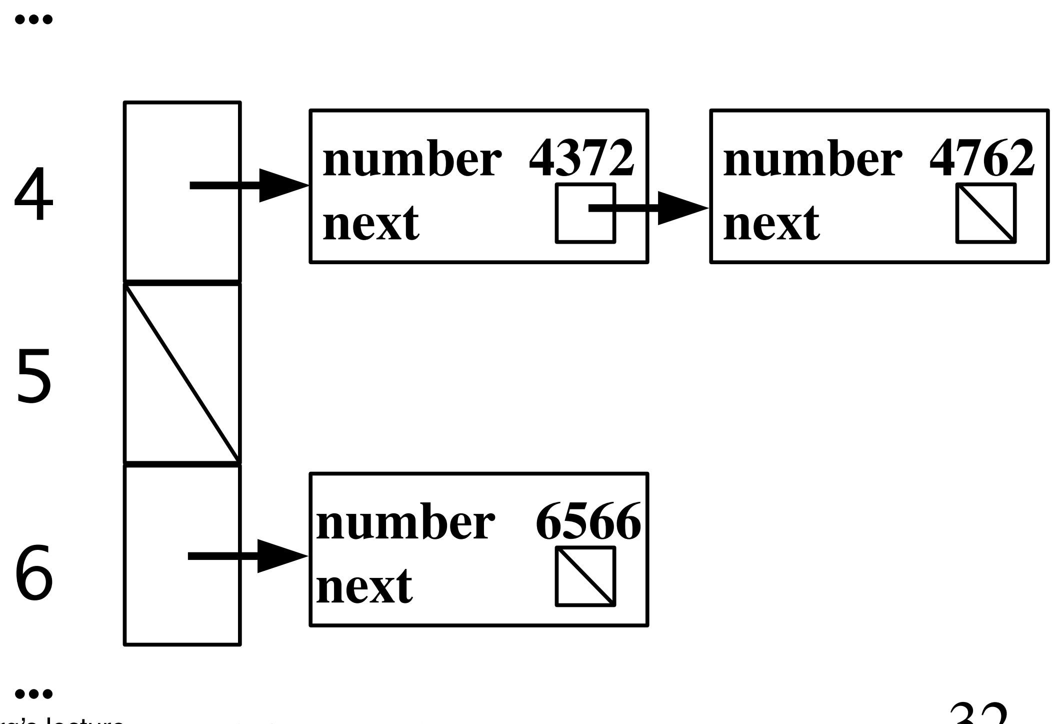
- What if numbers not random, eg likely to be near each other?
  - convert n to index in some other way, e.g. index = n mod 500
  - In general, function that makes each index equally likely: “makes hash out of any pattern in the numbers” -
- Hash function: converts data to hash code
- Mapping function: converts hash code to array index. (Why separate this?)

Collisions

- Even with 500 indices for 10 numbers, it is possible that more than one number will hash to same index
- As we reduce number of indices probability of collision grows
- => must be some way to handle collisions

Chaining

- **Instead of an array index referring to a single object, have it refer to a linked list of objects.**



Complexity

- Worst case: O(n)
  - all items hash to same index
- Average: depends on load factor alpha = n / size

alpha	Average compares
.1	1.05
.5	1.3
.8	1.4
.9	1.45
. 99	1.5

Built-in Hashing in Java

- The class java.util.HashMap<K, V>
  - Mapping from (unique) key to a value
  - Note: generic with two class parameters:
    - K: class of keys
    - V: class of values
  - E.g. NetID => Student java.util.HashMap<NetID, Student>
  - See JDK API
  - See Driver.java, UseDriverMap.java