

Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 11

Hashtables, and wrapping up

Advice on the mini-project – how to explain code

γνωθι σαυτον

```
ptr = t
while ptr.left != None:
    ptr = ptr.left
```

gnothi safton

we first set `ptr` to `t` and then we keep setting `ptr` to `ptr.left` until `ptr.left` becomes `None`

know thyself

we find the minimum element of the subtree of the BST starting at node `t`, by using a pointer `ptr` that starts from `t` and goes left until the next left child is `None`

Data structures for storing and retrieving

We have so far seen several data structures that can be used for storing elements of some type.

the best overall
complexity is that
of BSTs

binary search tree (ordered)
add, search, remove are all $O(\log n)$ (if tree is balanced)
if tree is not balanced: $O(n)$

n is the total size,
i.e. the total number
of elements

array and count	
<i>unordered</i>	<i>ordered</i>
add an element, i.e. append, is $O(1)$ (most times)	add an element: - find its position is $O(\log n)$ - insert it is $O(n)$
searching an element is $O(n)$	search an element is $O(\log n)$ (binary search)
removing an element is $O(n)$	as with add, this takes $O(n)$ overall

linked list (unordered)
add an element is $O(1)$ (add at head position)
searching an element is $O(n)$
removing an element is $O(n)$

Can we do better than $\log n$?

For example, what is the best way to store numbers,
if we **know** that they are all between 0 and 99?

0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2

Just use an array of size 100 to store their multiplicities!

1	0	2	0	1	1	0	0	0	0	1	0	...	0	1
---	---	---	---	---	---	---	---	---	---	---	---	-----	---	---

What is the complexity of add, search or remove?

- it is $\Theta(1)$, i.e. constant time!

this is because we just need to read or write a **known** position of the array
(i.e. to see if 4 is stored, we just read the 5th element of the array and
check if it is greater than 0 – we don't need to search)

Can we generalise this to arbitrary data?

For example, what is the best way to store numbers,

if we **don't know** whether they are all between 0 and 99?

0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103 ?

We can resize our array to a size > 2103 :

- this would work but would be very space-inefficient – in this case, all positions after 99 and before 2013 would be redundant
- to make this more evident, in order to store the number 1 million, we need to have an array of 1 million places (~1MB)

Also, what if we want e.g. to store strings instead of numbers?

Can we generalise this to arbitrary data?

For example, what is the best way to store numbers,

if we **don't know** whether they are all between 0 and 99?

0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103 ?

Idea:

- we can store the numbers by finding their **modulo** by 100
- and store in the array for each number: the number & its multiplicity

i.e. the remainder
after dividing
them by 100

0,1	-	2,2	2103,1	4,1	5,1	-	-	-	-	10,1	-	...	-	99,1
-----	---	-----	--------	-----	-----	---	---	---	---	------	---	-----	---	------

What is the complexity of add, search or remove? Still $\Theta(1)$!

Is this going to work? How can we now add 103?

And what if we want e.g. to store strings instead of numbers?

Hash tables

Hash tables are a data structure that allows us to store elements of arbitrary type in constant time using these ideas:

- store elements in an array
- use a constant-time function to compute for each element its index in the array – this is called a **hash function**
- use some smart solution to **resolve collisions**
-> collision is when two or more elements end being assigned the same index (e.g. 2, 102, 202, ... in example)

Hash tables with linked lists

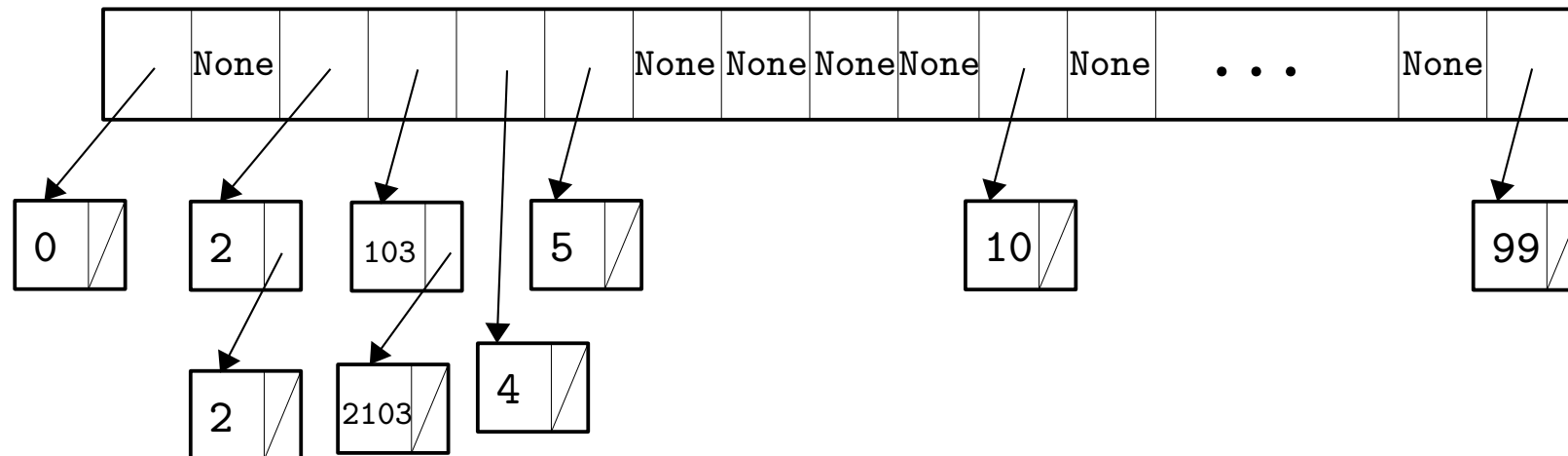
The standard solution to resolve collisions is to use **linked lists**!

That is, in each cell of the array we store a linked list of all the elements that are mapped to that cell.

So, in our previous example:

0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103, 103

we can store the elements in a hashtable like the following – note that we do not need multiplicity counters now:



*note: multiplicity counters are always neat and good to have and they go well with hash tables – here we did not include them because we want to demonstrate hash tables in their simplest form

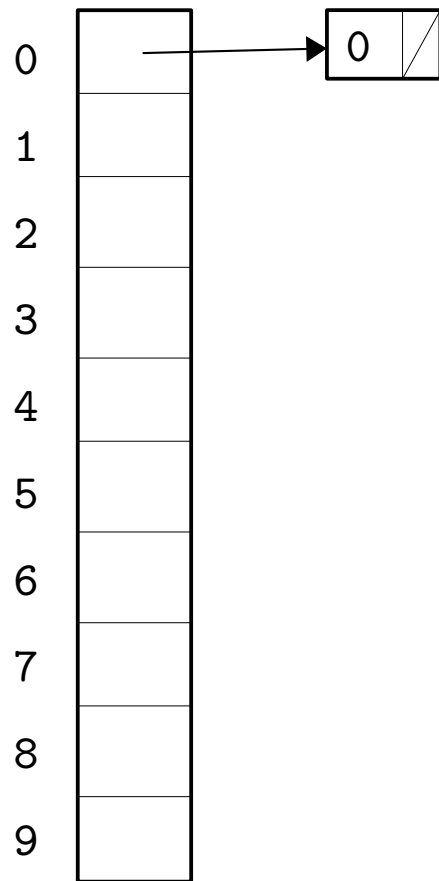
Example hash table operations

Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Example hash table operations

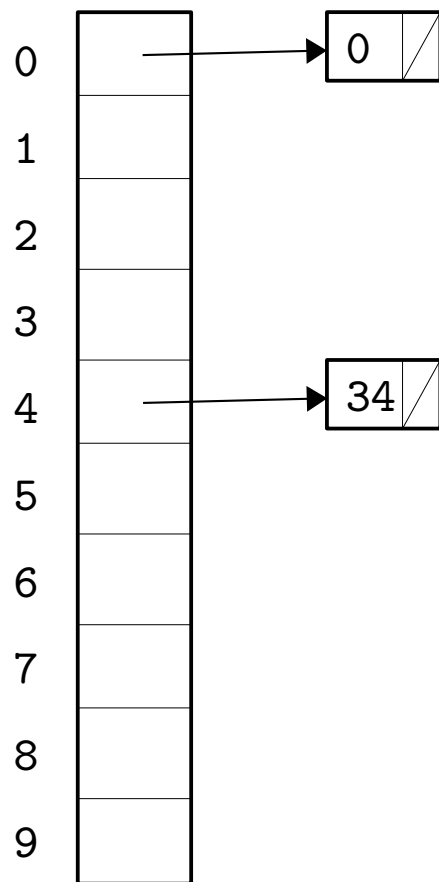
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0

Example hash table operations

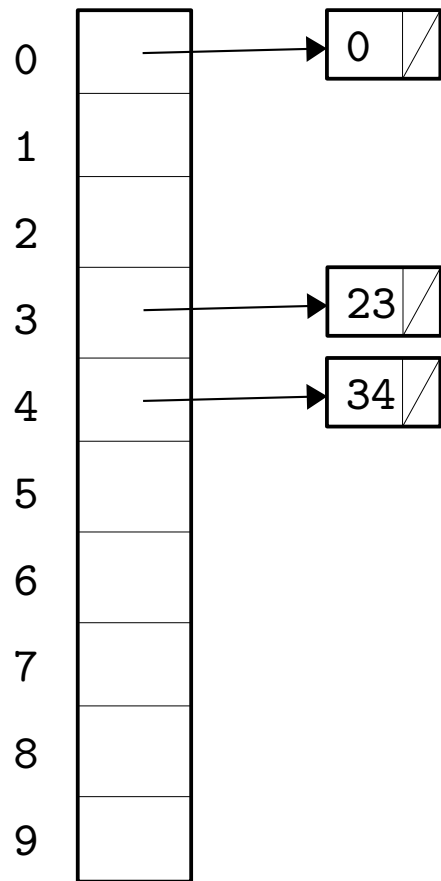
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4

Example hash table operations

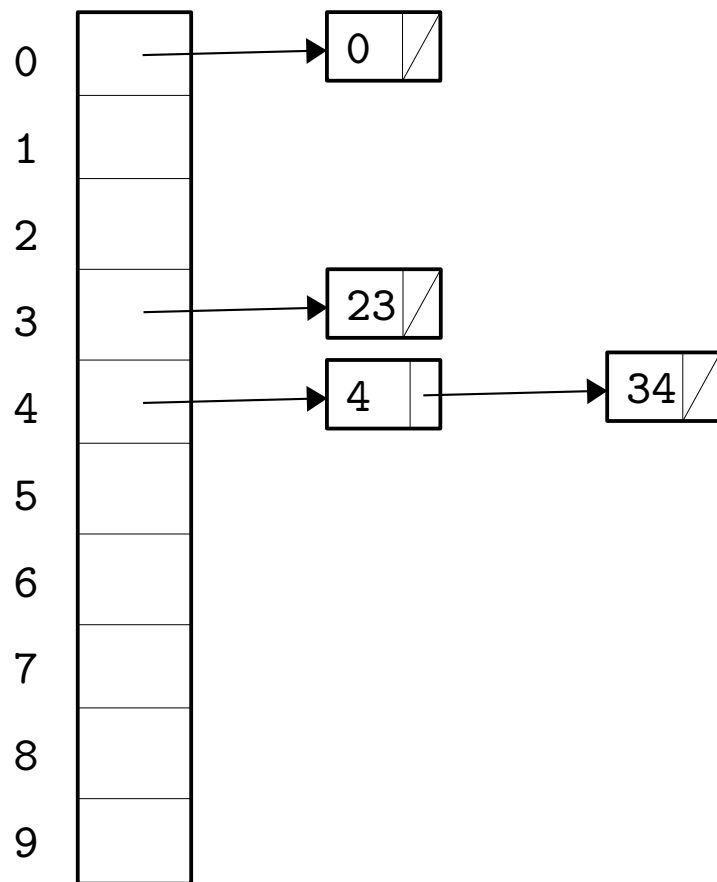
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3

Example hash table operations

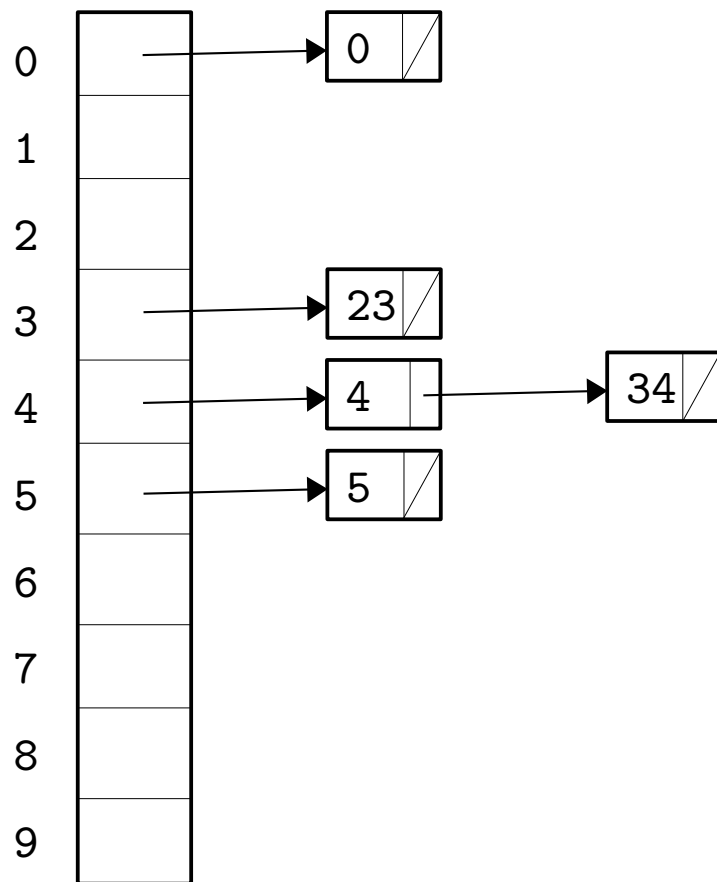
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list

Example hash table operations

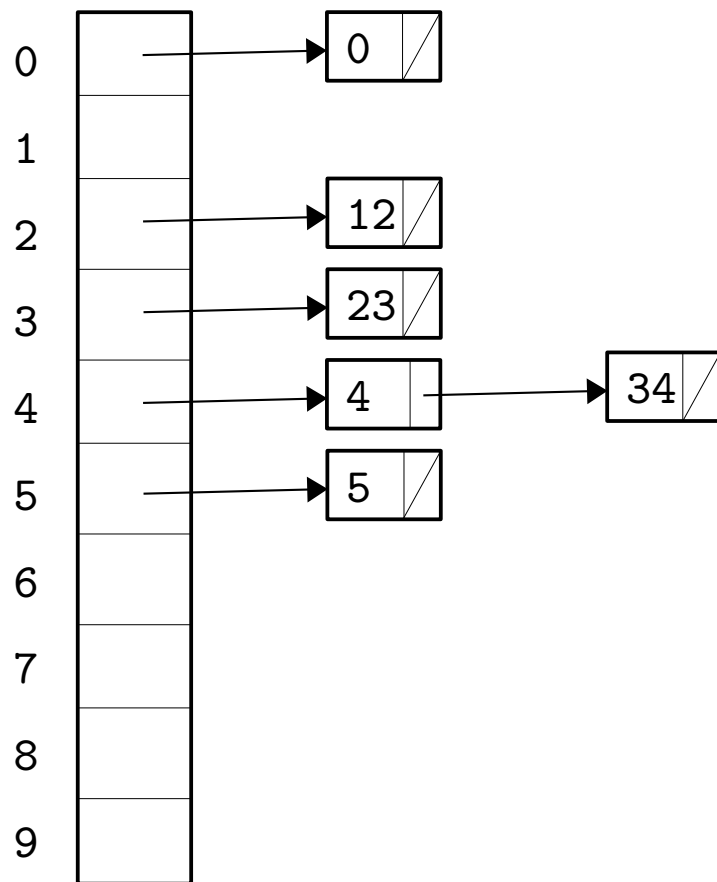
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5

Example hash table operations

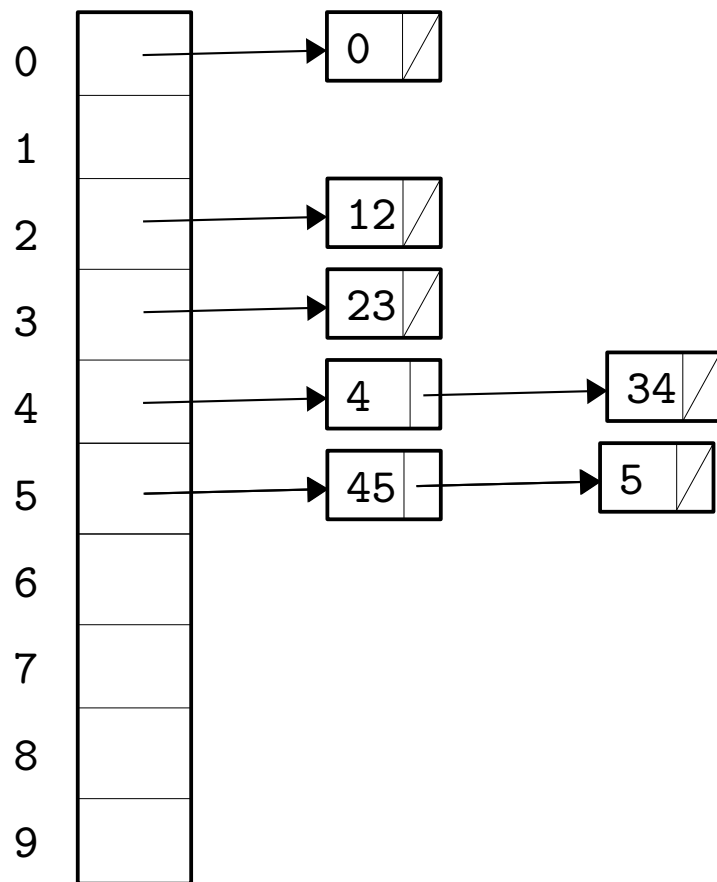
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2

Example hash table operations

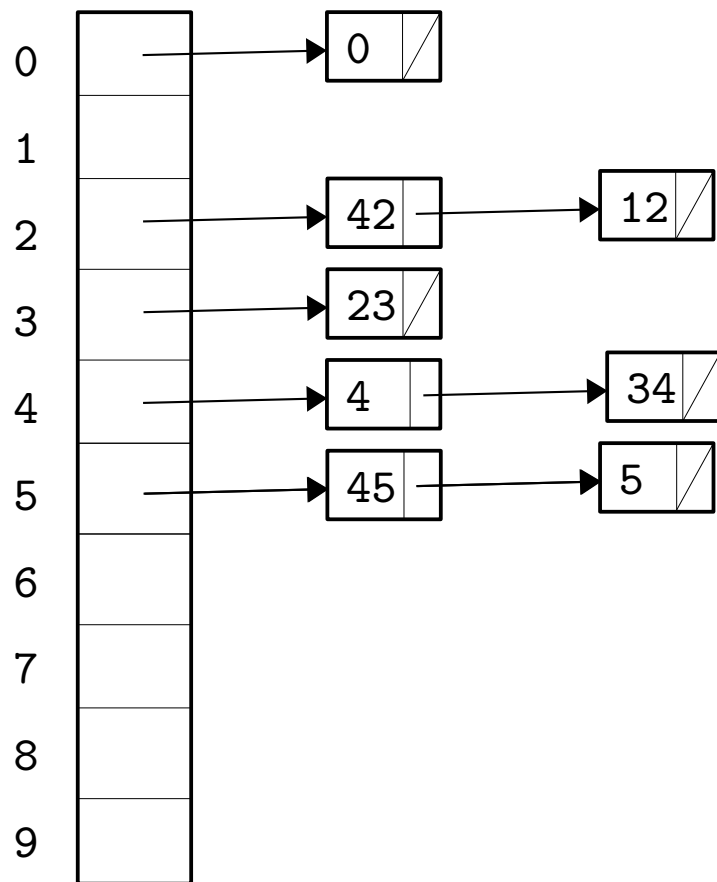
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)

Example hash table operations

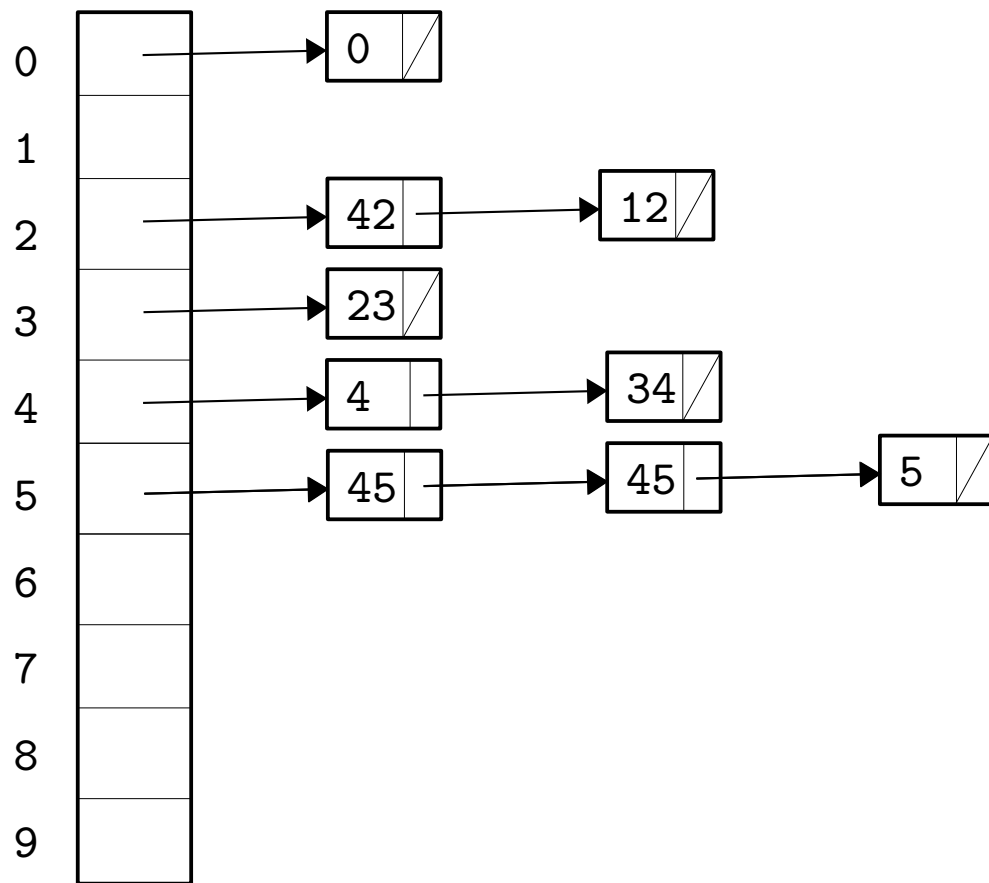
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)

Example hash table operations

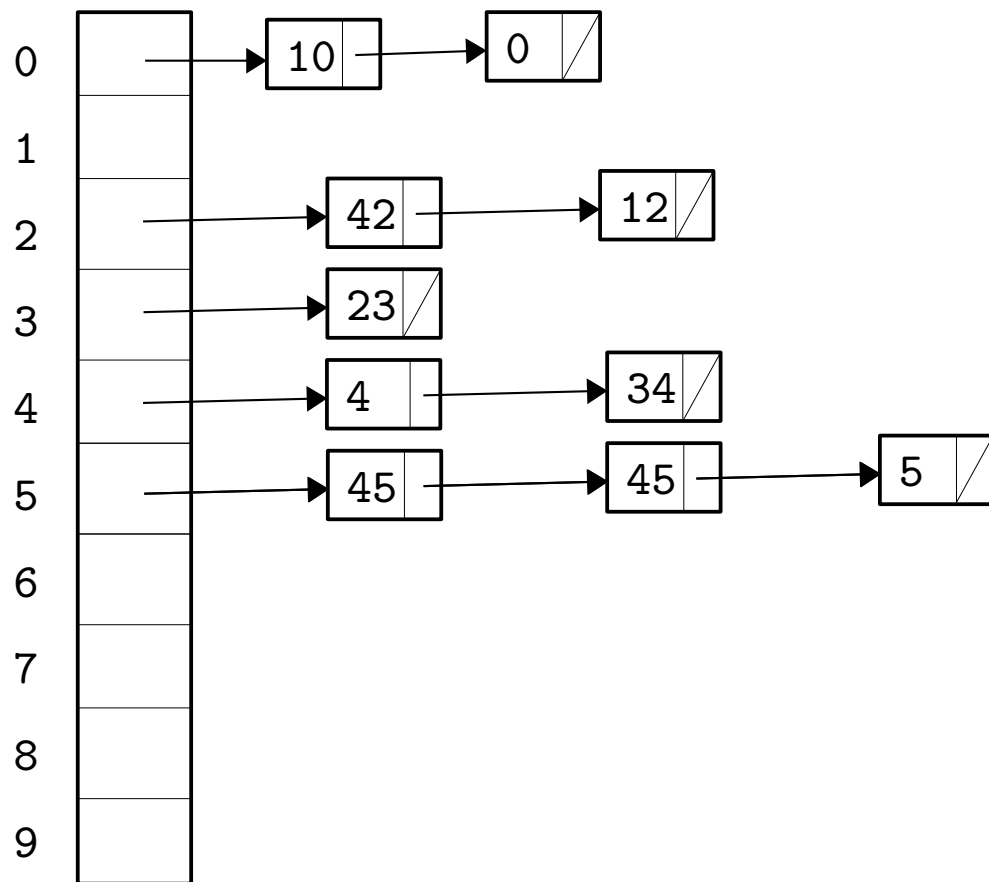
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)

Example hash table operations

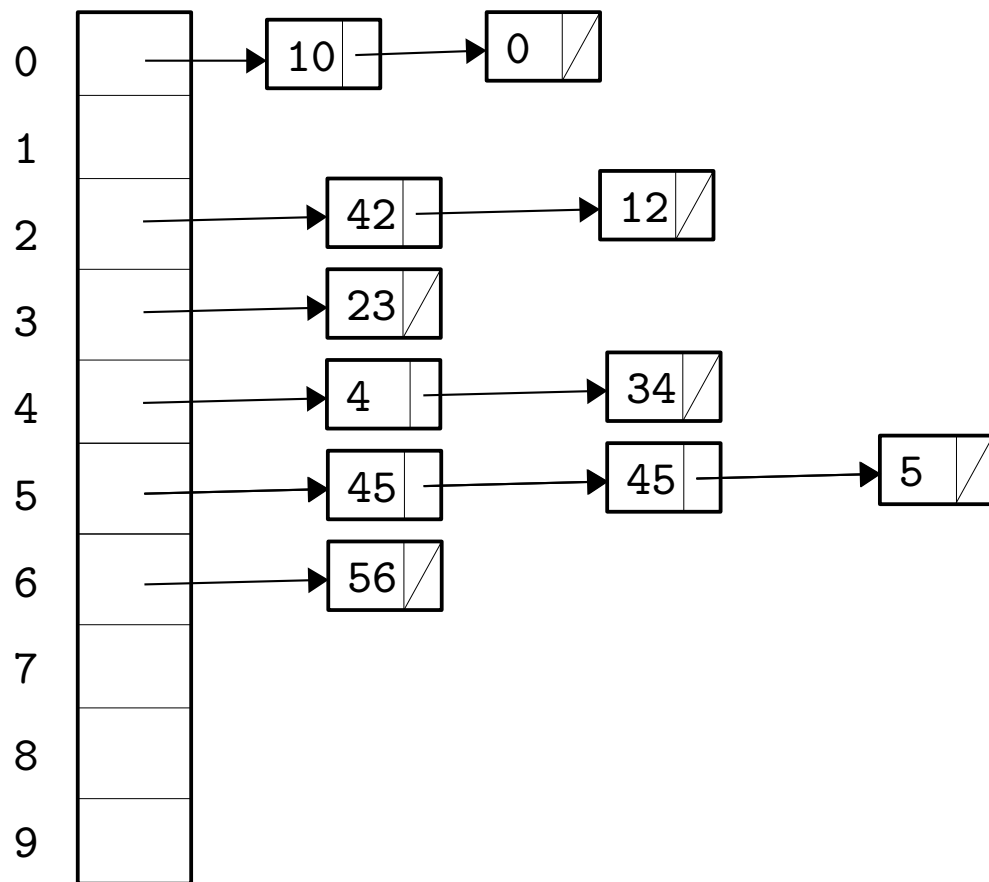
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)

Example hash table operations

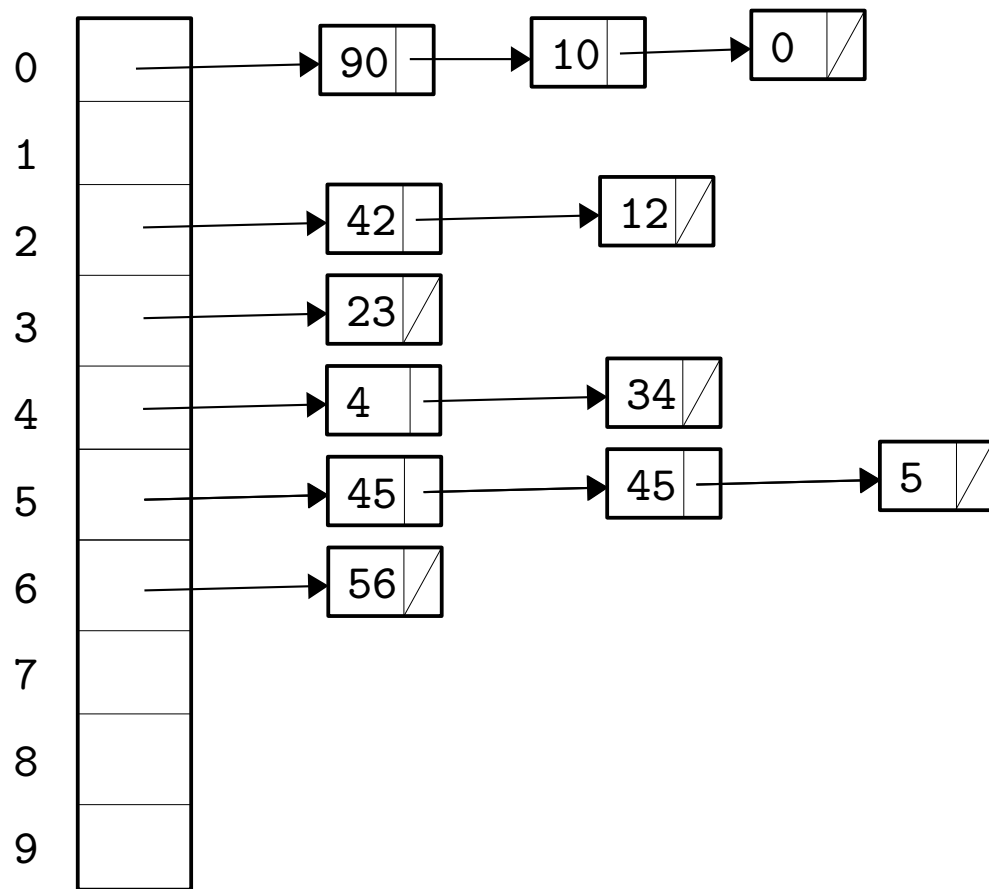
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6

Example hash table operations

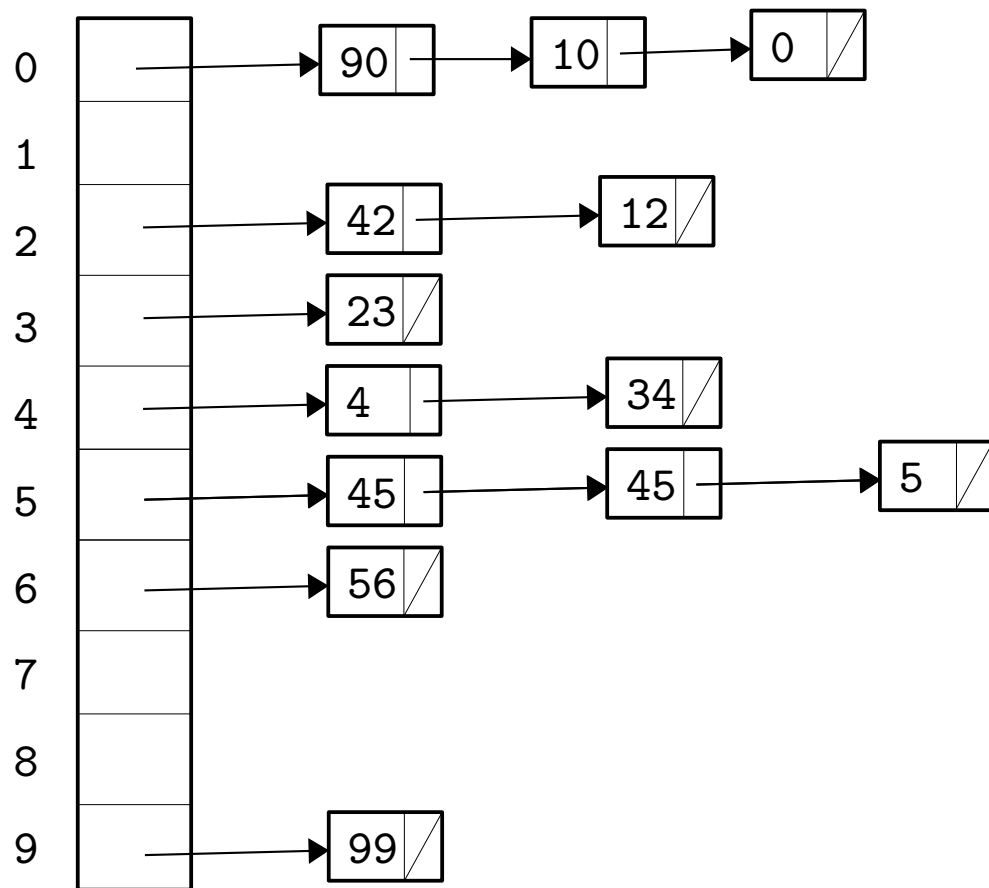
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6
- then, 90 is inserted in position 0 (at head)

Example hash table operations

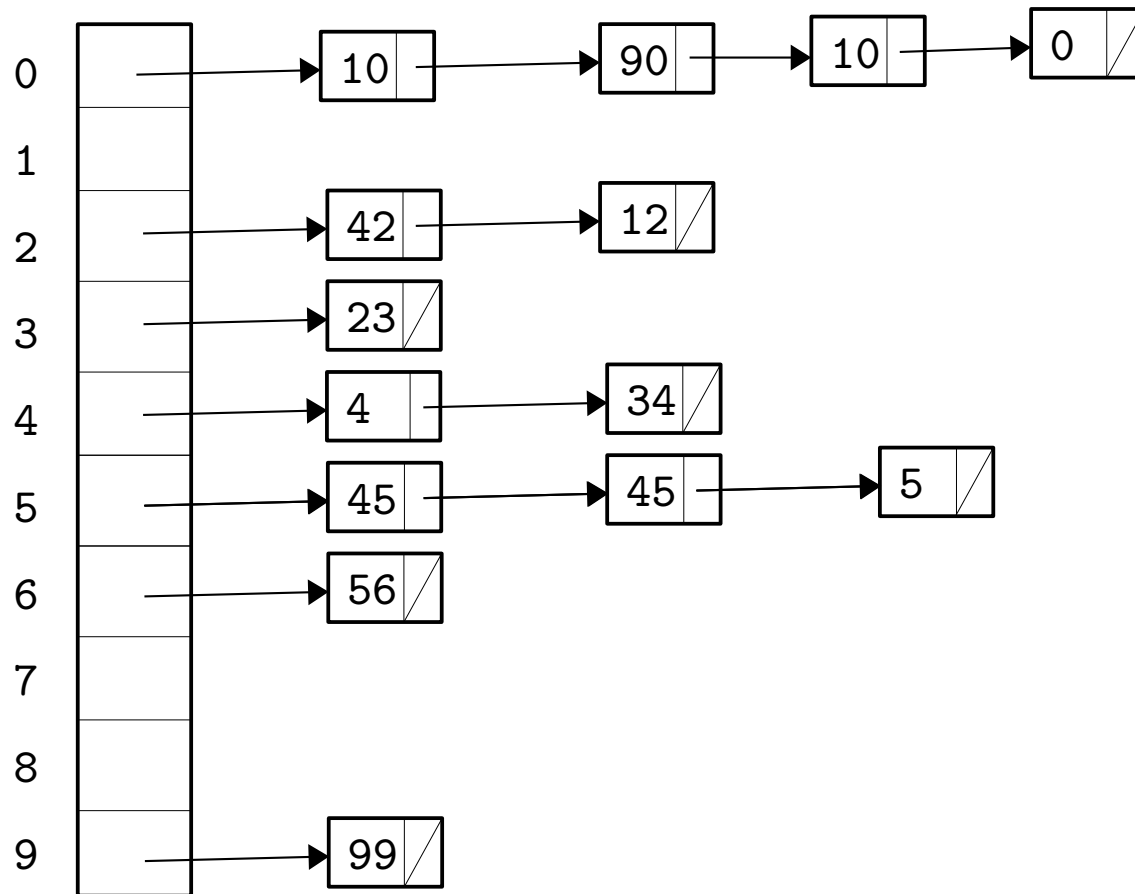
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6
- then, 90 is inserted in position 0 (at head)
- then, 99 is inserted in position 9

Example hash table operations

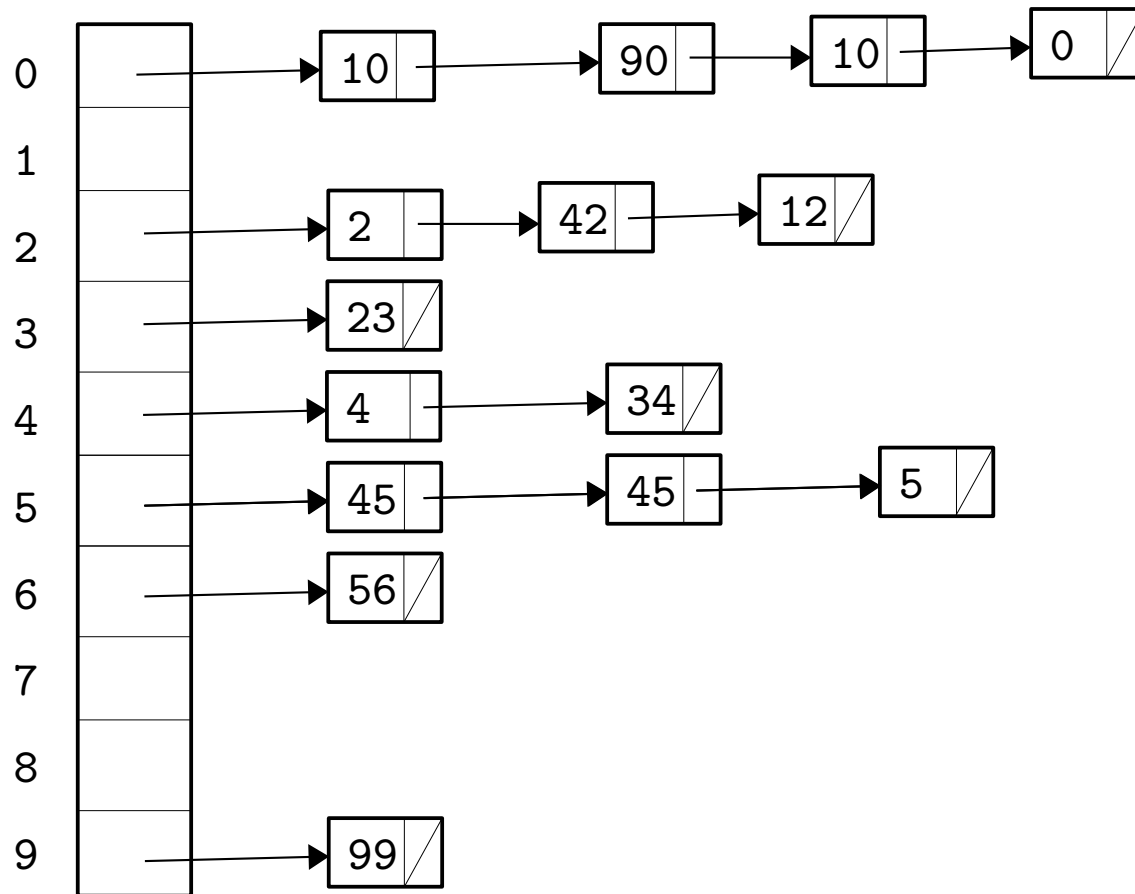
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6
- then, 90 is inserted in position 0 (at head)
- then, 99 is inserted in position 9
- then, 10 is inserted in position 0 (at head)

Example hash table operations

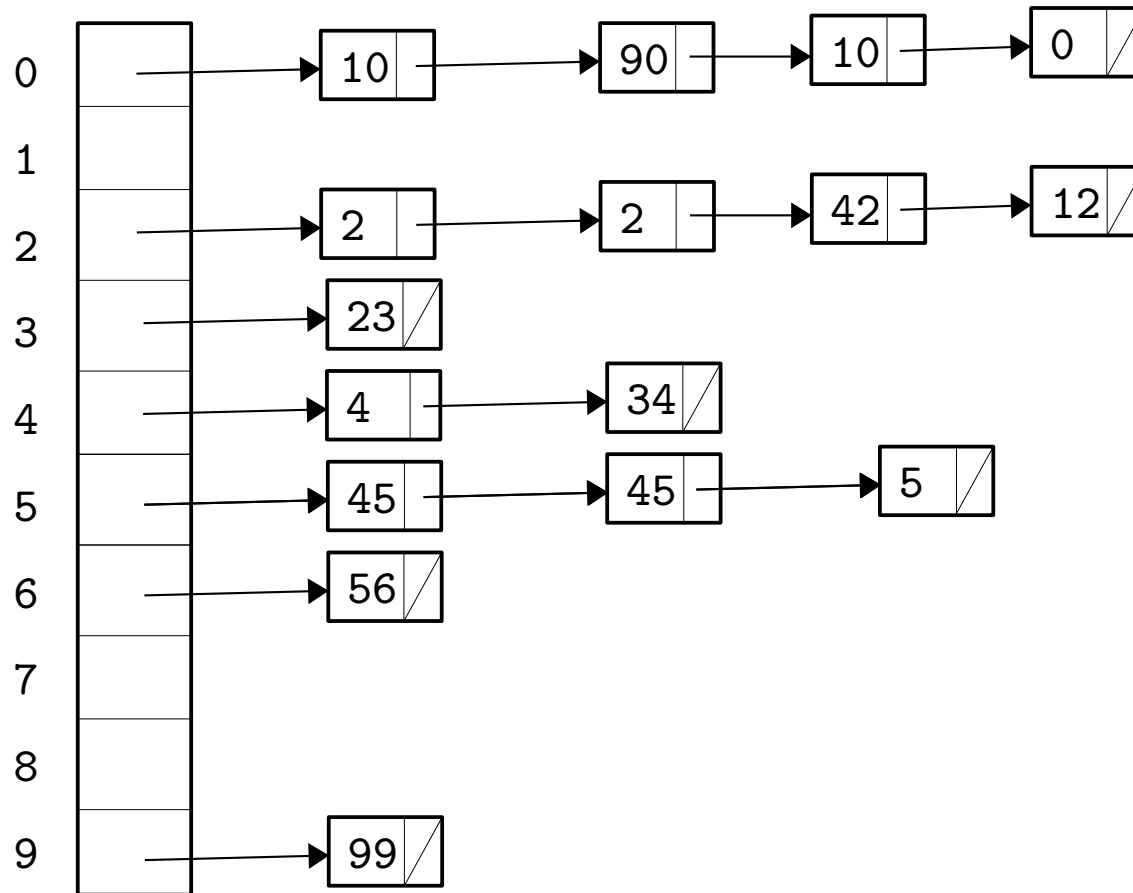
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6
- then, 90 is inserted in position 0 (at head)
- then, 99 is inserted in position 9
- then, 10 is inserted in position 0 (at head)
- then, 2 is inserted in position 0 (at head)

Example hash table operations

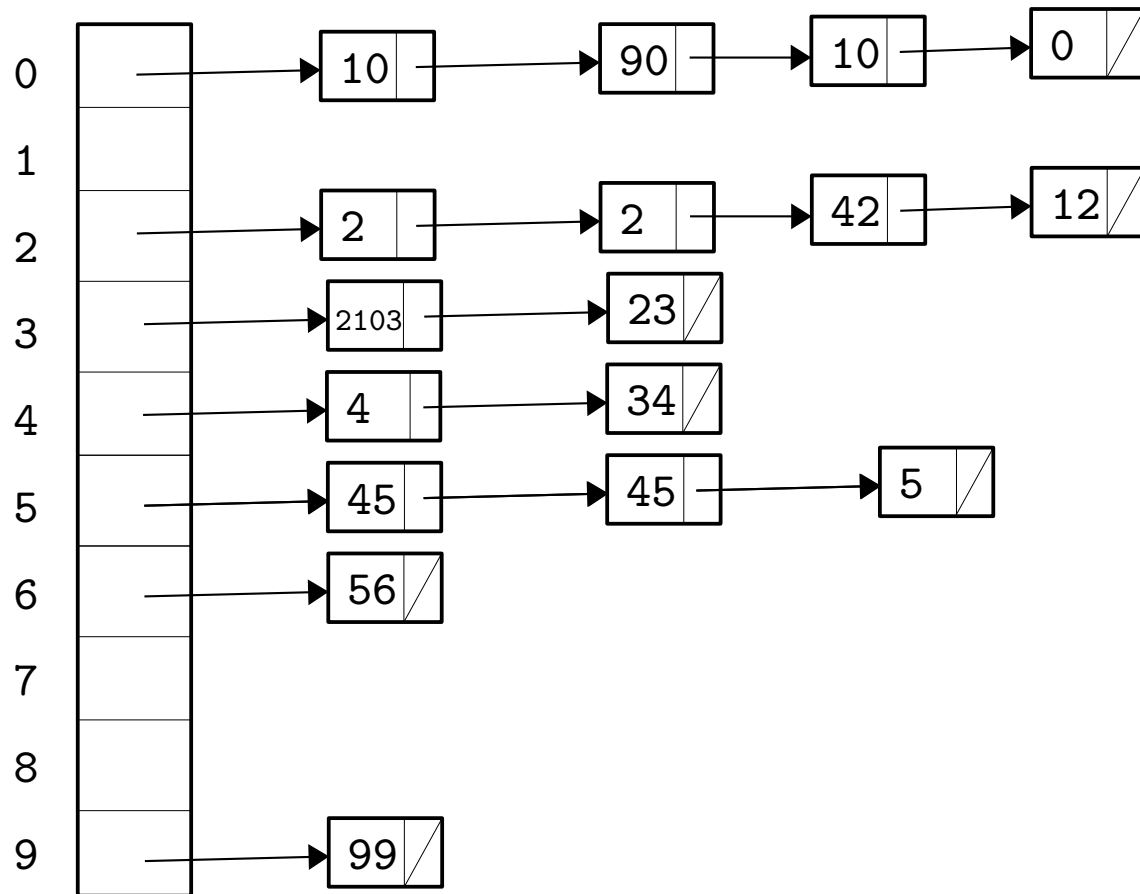
Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6
- then, 90 is inserted in position 0 (at head)
- then, 99 is inserted in position 9
- then, 10 is inserted in position 0 (at head)
- then, 2 is inserted in position 0 (at head)
- then, 2 is inserted in position 0 (at head)

Example hash table operations

Let us consider a hash table of size 10 (for simplicity). It is initially empty, and we then add: 0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103



- first, 0 is inserted in position 0
- then, 34 is inserted in position 4
- then, 23 is inserted in position 3
- then, 4 is inserted in position 4, at the head of the list
- then, 5 is inserted in position 5
- then, 12 is inserted in position 2
- then, 45 is inserted in position 5 (at head)
- then, 42 is inserted in position 2 (at head)
- then, 45 is inserted in position 5 (at head)
- then, 10 is inserted in position 10 (at head)
- then, 56 is inserted in position 6
- then, 90 is inserted in position 0 (at head)
- then, 99 is inserted in position 9
- then, 10 is inserted in position 0 (at head)
- then, 2 is inserted in position 0 (at head)
- then, 2 is inserted in position 0 (at head)
- and, finally, 2013 is inserted in position 3 (at head)

Implementation of hash tables

There are two data structures involved in hash tables:

- an array
- a linked list, for each element of the array

The functions we need to implement are:

- one for creating an empty hash table
- one for adding an element in a hash table
- one for searching an element in a hash table
- one for removing an element in a hash table
- a hash function, i.e. one that takes arbitrary integers and maps them to an index inside the array bounds
(this was the modulo function before, but there are many choices)

Initialisation of hash tables

We implement hash tables with array size 10 (i.e. as last example).

To initialise a hash table we create an array, which we initialise to some default size (here we pick 10)

We also set the size of the hash table to 0 (this counts how many elements we have stored).

```
class HashTable:
    def __init__(self):
        self.inArray = [LinkedList() for i in range(10)]
        self.size = 0
```

Adding to a hashtable

To add an element d , we need to do 3 things:

- find the index of d , i.e. $\text{hash}(d)$
- add d in the linked list that is in position
- increase the size of the hash table by 1

```
def hash(self, d):  
    return d % len(self.inArray)  
  
def add(self, d):  
    i = self.hash(d)  
    self.inArray[i].insert(0,d)  
    self.size += 1
```

Searching and removing

These work in the same way as adding, i.e. by finding the hash of the element we want to search/remove, and then use the LinkedList operations at that position to do the search/remove:

```
def search(self, d):  
    i = self.hash(d)  
    if self.inArray[i].search(d) == -1: return False  
    return True  
  
def remove(self, d):  
    i = self.hash(d)  
    oldLength = self.inArray[i].length  
    self.inArray[i].removeVal(d)  
    if self.inArray[i].length < oldLength:  
        self.size -= 1
```

Complexity considerations

What is the time complexity for add, search and remove?

Well, in all of them:

- finding the index (i.e. the hash) of an element is $\Theta(1)$
- finding the linked list at the indexed position is $\Theta(1)$

and then, things vary for each function:

- adding the element in the linked list is $\Theta(1)$
- searching or removing the element from the linked list is $\Theta(l)$ in the worst case, where l is the length of the linked list

So, overall, in the worst case:

- add is $\Theta(1)$
- search and remove are $\Theta(l_{\max})$, where l_{\max} is the length of the longest linked list

Linked list length

So, the most important factor here is: how long can the linked lists get?

- in theory, they can be as long as the size of the hash table
- this happens if all (or most of) the elements are put in one list (e.g. if array has size 10 and we add numbers with the same last digit)
- complexity in this case is horrible:
 - if the hash table has size n (i.e. contains n elements) then search and remove have complexity $\Theta(n)$ in the worst case!

How can we make sure that linked lists do not get that long?

We can do 2 things:

- use a *good hash function and array size* that spread elements across the array
- *resize up* the array once too many elements are stored (so, collisions become inevitable)

Hash functions and array length

To find where to store an integer i in an array of size 10 we did:

$$\text{hash}(i) = i \% 10$$

where the % function finds the remainder of dividing its first argument by its second argument.

What if e.g. most of the numbers we want to store end in 3?

We can do 2 things:

- **use an array of a different size**. E.g. if size = 11, this spreads things out:
 - $3 \% 11 = 3$, $13 \% 11 = 2$, $23 \% 11 = 1$, $33 \% 11 = 0$, $43 \% 11 = 10$, etc.
- **modify the hash function** so that it is not just %. E.g. if we use:

$$\text{hash}(i) = ((i // 10) + (i \% 10)) \% 10$$

then the hash also depends on the second digit of each number:

- $((3 // 10) + (3 \% 10)) \% 10 = 3$, $((13 // 10) + (13 \% 10)) \% 10 = 4$,
 $((23 // 10) + (23 \% 10)) \% 10 = 5$, $((33 // 10) + (33 \% 10)) \% 10 = 6$, etc.

Uniform hashing

By choosing a hash function that spreads elements uniformly among the array, we can achieve the **optimal** worst-case complexity.


By spreading uniformly we mean that each element has equal probability to be assigned to any of the slots (cf. uniform distribution).

This is called **simple uniform hashing**.

So, if we assume that we have simple uniform hashing:

- if the hash table has n elements and its internal array has length m
- then average length of each linked list is going to be n/m
- so, complexities:
 - add: $\Theta(1)$ in the worst case (just add at the head of the list)
 - search and remove: $\Theta(n/m)$ in the worst case (we may need to examine a whole linked list to find our element)

sometimes called
the **load factor** of
the hash table



Resizing up

So, overall, the hash table operations have complexity $O(n/m)$, where:

- n is the size of the hash table and m is the length of its internal array

As we add elements to the hash table:

- the ratio n/m increases
- the overall performance decreases (search and remove take longer)

There is a practical solution to this:

*whenever the ratio n/m gets above a **constant threshold**,
we **resize up** the internal array*

In this way, the overall complexity is $O(\text{threshold})$, i.e. $O(1)$. *

* resizing up takes $O(n)$, so we only achieve $O(1)$ most of the time (on average)

The threshold is determined by the user/implementor of the hash table.

E.g. in Java standard Hashtable, it is set to 0.75.

Adding to a hashtable – with resizing up

To add an element d , we need to do 4 things:

- ... (as before) + if the size goes beyond the threshold, we resize up

```
class HashTable:
    def __init__(self):
        # default initial array length: 10, threshold: 0.75
        self.inArray = [LinkedList() for i in range(10)]
        self.size = 0
        self.threshold = 0.75

    def add(self, d):
        i = self.hash(d)
        self.inArray[i].insert(0,d)
        self.size += 1
        if self.size > self.threshold*len(self.inArray):
            self._resizeUp()
```

How to resize up

To resize up the hash table, we need to do 2 things:

- create a new array with larger length (e.g. double)
- re-insert all the elements from the old array to the new one, making sure we **re-hash** every element (the hash function has changed!)

```
def _resizeUp(self):
    oldArray = self.inArray
    self.inArray = [LinkedList() for i in range(2*len(oldArray))]
    for i in range(len(oldArray)):
        while oldArray[i].length > 0:
            d = oldArray[i].remove(0)
            self.inArray[self.hash(d)].insert(0,d)
```

Hash tables everywhere – hash functions

To store integers, we simply hashed them (using the mod operation), found their position in the array and inserted them in the corresponding linked list.

What if we want to store other things, not just integers?

In general, a **hash function** h takes values (of the kind that we want to store) and returns integers in the range: $0, 1, \dots, m-1$.

This can be done in several ways, for example it can be done in 2 stages:



We know how to hash integers, so we just need to figure out how to transform arbitrary values into integers:

- for strings, we can take their ASCII representation
- for arbitrary objects, their memory reference, etc.

What is important: **equal values always have equal hashes**

In practice

In practice, each programming language has some built-in construct to transform arbitrary values into integers, e.g. in Python:

```
st1 = "123"; st2 = "123"; st3 = "1234"  
print(hash(st1),hash(st2),hash(st3))
```

```
i1 = 123; i2 = 123; i3 = 1234  
print(hash(i1),hash(i2),hash(i3))
```

```
l1 = LinkedList(); l2 = LinkedList()  
print(hash(l1),hash(l2))
```

Data structures for storing and retrieving – updated

hash table (unordered)
add, search, remove are all $O(1)$ (most times) (if hashing is uniform)
if hashing is not uniform: $O(n)$

binary search tree (ordered)
add, search, remove are all $O(\log n)$ (if tree is balanced)
if tree is not balanced: $O(n)$

array and count	
<i>unordered</i>	<i>ordered</i>
add an element, i.e. append, is $O(1)$ (most times)	add an element: - find its position is $O(\log n)$ - insert it is $O(n)$
searching an element is $O(n)$	search an element is $O(\log n)$ (binary search)
removing an element is $O(n)$	as with add, this takes $O(n)$ overall

linked list (unordered)
add an element is $O(1)$ (add at head position)
searching an element is $O(n)$
removing an element is $O(n)$

From hash tables to hash maps

A **map** (or **dictionary**) is a data structure where we can store pairs of the form:

(key, value)

for some type of keys and values. For example:

- strings and their multiplicity (cf. mini-project)
- strings and their definitions (as in dictionaries)
- x 's and $f(x)$'s (for an arbitrary function f), etc.

This is easy with hash tables:

- we build hash tables that store key-value pairs
- to store a pair, we hash its key to find its position in the hash table
- we make sure that what we implement is a **map**: adding the same key twice overwrites its previous value

We call the resulting data structure a **hash map**.

In Python

In Python, hash maps are built-in data structures:

```
m = {}                                # builds an empty dictionary (hash map)
print(m, type(m))
m["foo"] = "a standard name for variables"    # add (key,value)
m["flood"] = "a fairly large amount of water"
m["algorithms"] = "data structures"
print(m)
m["algorithms"] = "go well with data structures" # value update
print(m)
```

More at: <https://docs.python.org/3/tutorial/datastructures.html#dictionaries>

How to use hash tables and maps

Hash tables and hash maps are the typical choice of data structure for storing and retrieving data (especially lots of them) in an efficient way.

For example, remember our dynamic programming algorithms:

```
def fibDP(n):  
    memo = [-1 for i in range(n+1)]  
    return fibMem(n,memo)  
  
def fibMem(n, memo):  
    if memo[n] != -1: return memo[n]  
    if n <= 1: memo[n] = n  
    else: memo[n] = fibMem(n-1,memo)+fibMem(n-2,memo)  
    return memo[n]
```

How to use hash tables and maps

Hash tables and hash maps are the typical choice of data structure for storing and retrieving data (especially lots of them) in an efficient way.

With dictionaries (i.e. hash maps):

```
def fibDP(n):  
    memo = {}  
    return fibMem(n, memo)  
  
def fibMem(n, memo):  
    if n in memo: return memo[n]  
    if n <= 1: memo[n] = n  
    else: memo[n] = fibMem(n-1, memo) + fibMem(n-2, memo)  
    return memo[n]
```

However, using an array for memo was a reasonable (i.e. efficient) choice in this case.

How to use hash tables and maps

Here instead, using a double array for memo can be a waste of memory:

```
def coinSplitDP(m):  
    memo = [[-1 for j in range(len(coin))] for i in range(m+1)]  
    return coinSplitMem(m,0,memo)  
  
def coinSplitMem(m, startCoin, memo):  
    # ...
```

Much of the array will remain empty. With hash maps, we would only be storing the elements we need – left as exercise!

- we would be using pairs $(m, \text{startCoin})$ as keys
- and integers (representing min number of coins) as values

In general, when our data cannot be stored and retrieved efficiently with arrays, we can check if hash tables or hash maps can be used.

Summary

- Hash tables are an efficient way to store and retrieve data.
They are based on these ideas:
 - arrays are fast to access if we know the index we are looking for
 - we use a hash function to find an element's index
 - we store linked lists instead of single elements to resolve collisions
- They can store elements of arbitrary type, so long as we know how to hash them
- They can be used in order to implement efficient maps – *hash maps*

That's all for now!

Exercises

1. Similarly to what we saw in slides 9-26, starting from an empty hash table of size 15, and assuming a hash function:

$$\text{hash}(i) = i \% 15$$

draw the hashtable we obtain after inserting consecutively the elements:

0, 34, 23, 4, 5, 12, 45, 42, 45, 10, 56, 90, 99, 10, 2, 2, 2103

2. Explain what is the worst-case complexity of each of the hash table operations, in each of these cases:
 - a) if we assume simple uniform hashing and use resizing-up
 - b) if we assume simple uniform hashing and do not use resizing-up
 - c) if we do not assume simple uniform hashing and do not use resizing-up