# Data Structures and Algorithms

## Lesson 9: *Make a hash of it*

*Hash tables, separate chaining, open addressing, linear/quadratic probing, double hashing, HashSets and HashMaps*

# Outline

1. **Why Hash?**
2. Separate Chaining
3. Open Addressing
   - Quadratic Probing
   - Double Hashing
4. HashSet and HashMap

# Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents
- This is often referred to as **associative memory** structure
- A classical example is a telephone directory
  - ⋆ We look up a name
  - ⋆ We want to know the number
- What data structure should we use?

# Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations
- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$
  - ⋆ We will study binary search later
  - ⋆ Maintaining an ordered list is costly ($\Theta(n)$ for insertion)
- We could use a binary search tree
  - ⋆ Search is $\Theta(\log(n))$
  - ⋆ Insertion/deletion is $\Theta(\log(n))$

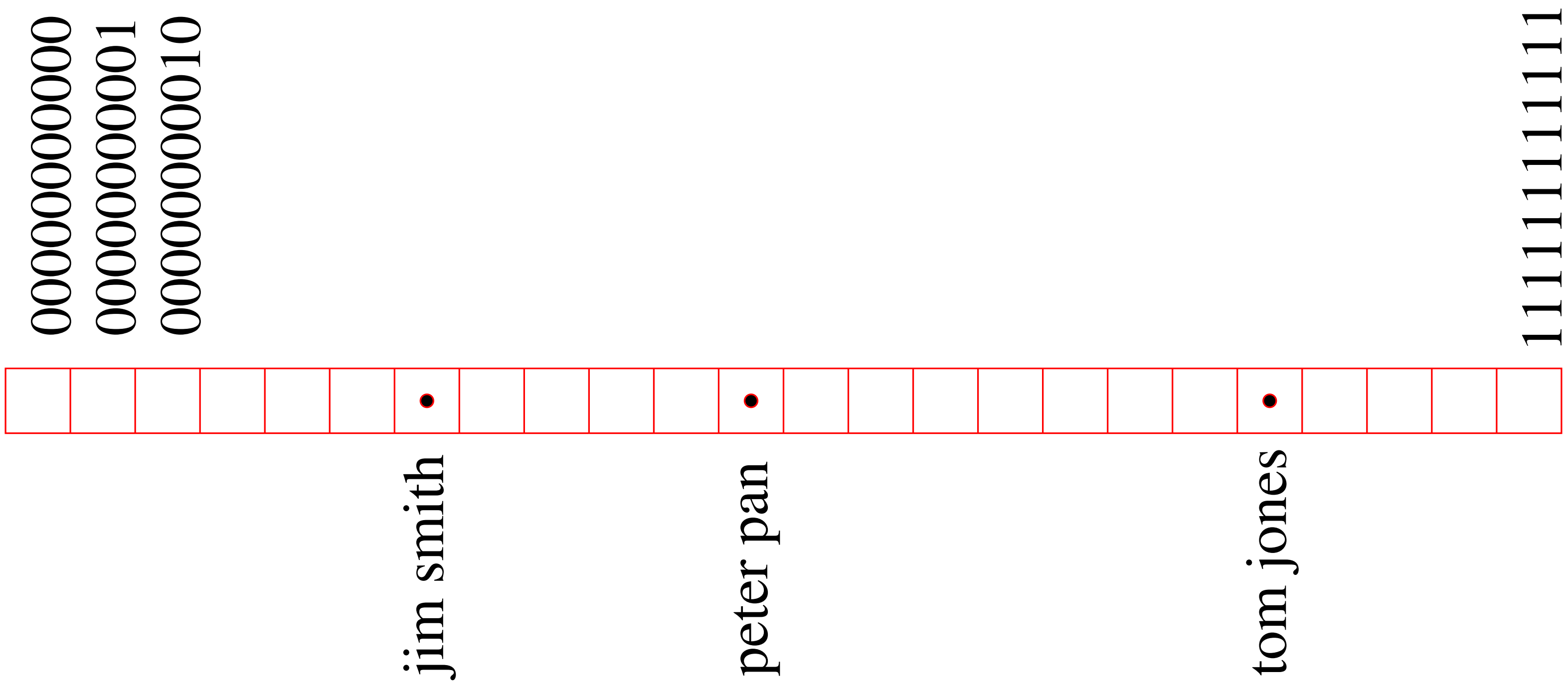
# Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions
- Let us consider the content we want to search on as a **key**
  - ⋆ For telephone numbers the key would be the name of the person we want to phone
- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array
  - ⋆ That is, the key is a string of, say, 100 characters, so can be represented by an 800 digit binary number
  - ⋆ We could look up the key in a table of $2^{800}$ items

# Hashing

- This approach is slightly wasteful of memory – almost all memory locations would be empty!

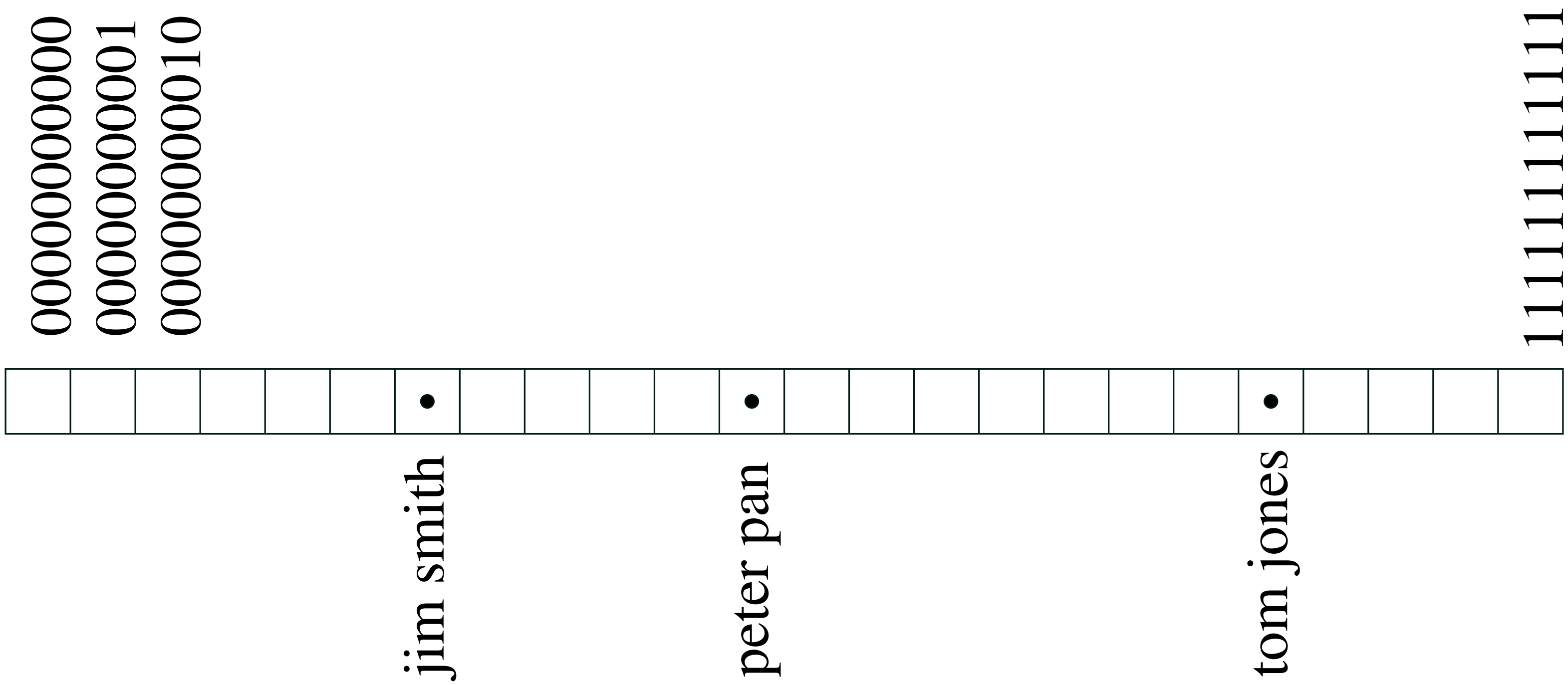
- We can save on memory by folding up the table up onto itself



# Hashing

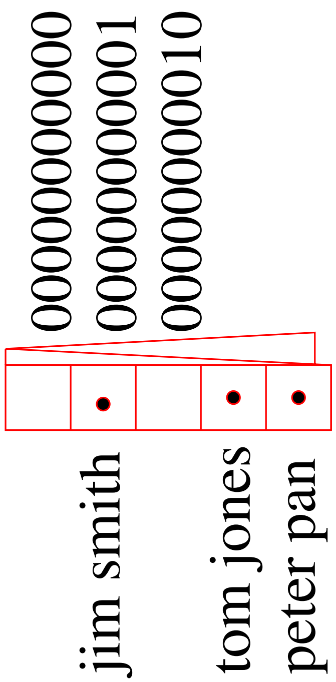
- This approach is slightly wasteful of memory – almost all memory locations would be empty!

- We can save on memory by folding up the table up onto itself



# Hashing

- This approach is slightly wasteful of memory – almost all memory locations would be empty!
- We can save on memory by folding up the table up onto itself



# Hashing Codes

- A **hashing function** `x.hashCode()` takes an object, `x`, and returns a positive integer, the **hash code**
- To turn the hash code into an address take the hash code modulo the table size

```
int index = Math.abs(x.hashCode() % tableSize);
```

- If `tableSize` $= 2^n$ we can compute this more efficiently using a mask

```
int index = Math.abs(x.hashCode() & (tableSize -1));
```

# Hashing Functions

- Hashing functions take an object and return an integer
- Java objects all inherit the method `hashCode()` from `Object`, although you will sometimes need to override this definition
- Hashing functions aren't magic
  - ⋆ They tend to add up integers representing the parts of the object
- We want similar objects to be mapped to different integers
- Sometimes two objects will be mapped to the same address
- Collision resolution is an important part of hashing

# What Makes a Good Hashing Function?

- fast to compute
- keys must be distributed as uniformly as possible to avoid collisions
- consistent with the equality testing function:
  - ⋆ equal objects have equal hashcodes
- good general-purpose hash function:

$$h(k) = k \,\%\, M$$

with $M$ a **prime** number used as table size

# Hashing Strings

- Java hashes Strings using a function

```
public static int hashCode()
{
  int h = 0;

  for(int i =0; i<s.length(); i++)
      h = 31*h + s[i];

  return h;
}
```

- The number 31 is to try to prevent clashes

# DIY

- All objects inherit from `Object` which has an in-built `hashCode()`
  - ⋆ the default `hashCode()` produces a code based on references – why?
- When you write your own class, you can use this but
  - ⋆ If you re-write `equals()` to implement a logical equality (rather than being the same reference) you *have to* implement a new `hashCode()` which is compatible
  - ⋆ In almost any application where you would want to use a hash table of objects you are likely to want to do this

# Outline

1. Why Hash?
2. **Separate Chaining**
3. Open Addressing
   - Quadratic Probing
   - Double Hashing
4. HashSet and HashMap

# Collision Resolution

- Collisions are inevitable and must be dealt with
- There are two commonly used strategies
  - ⋆ Separate chaining – make a hash table of lists
  - ⋆ Open addressing – find a new position in the hash table
- Collisions add computational cost
- They occur when the hash table becomes full
- If the hash table becomes too full then it may need to be resized

# Resizing a Hash Table

- Resizing a hash table is easy
  - ⋆ Create a new hash table of, say, twice the size
  - ⋆ Iterate through the old hash table adding each element to the new hash table
- Note that you have to recompute all the hash codes
- Resizing a hash table has a small amortised cost, but can give you a very hiccupy performance
- The size of a hash table is a classic example of a memory/space versus execution time trade off – using bigger (sparser) hash tables speeds up performance

# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

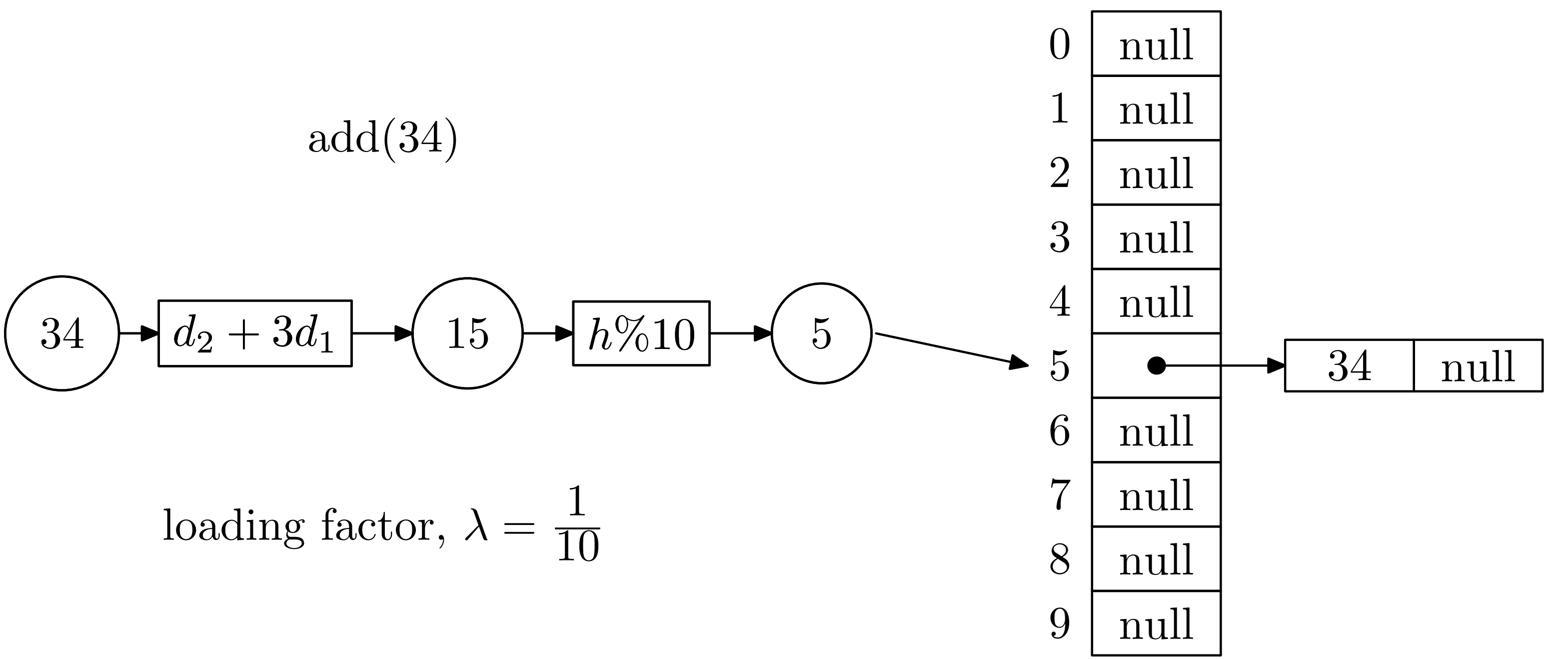
$$d_2d_1 \rightarrow \boxed{d_2 + 3d_1} \rightarrow h \rightarrow \boxed{h\%10} \rightarrow i$$

loading factor, $\lambda = \frac{0}{10}$

| | |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| 8 | null |
| 9 | null |

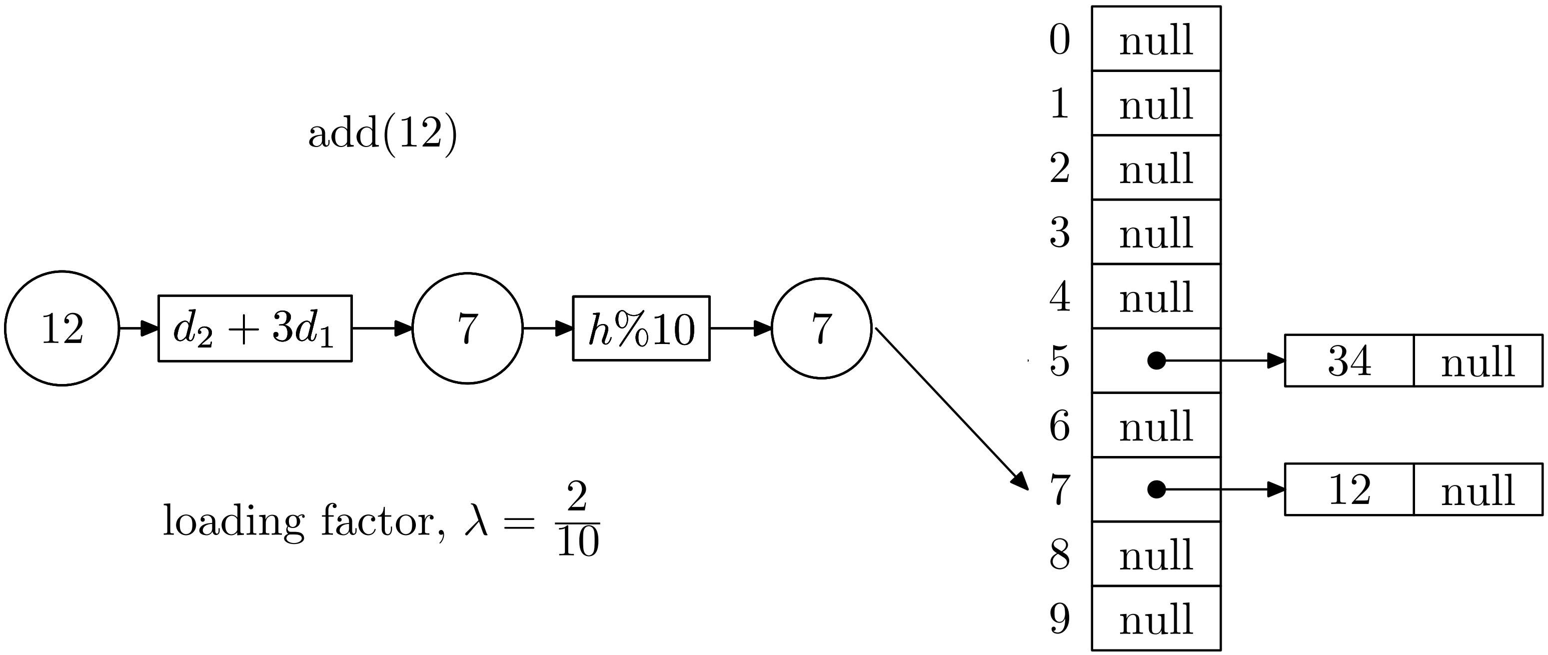
# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



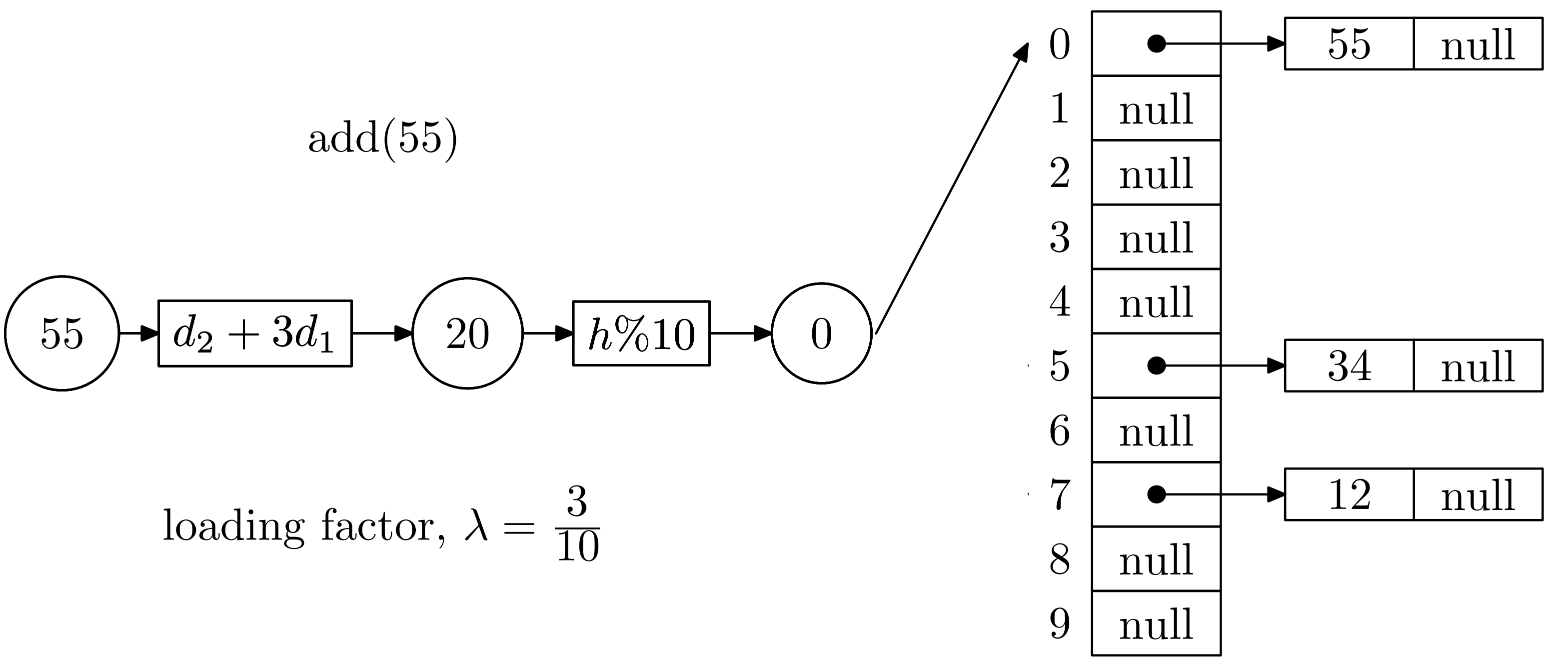
# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



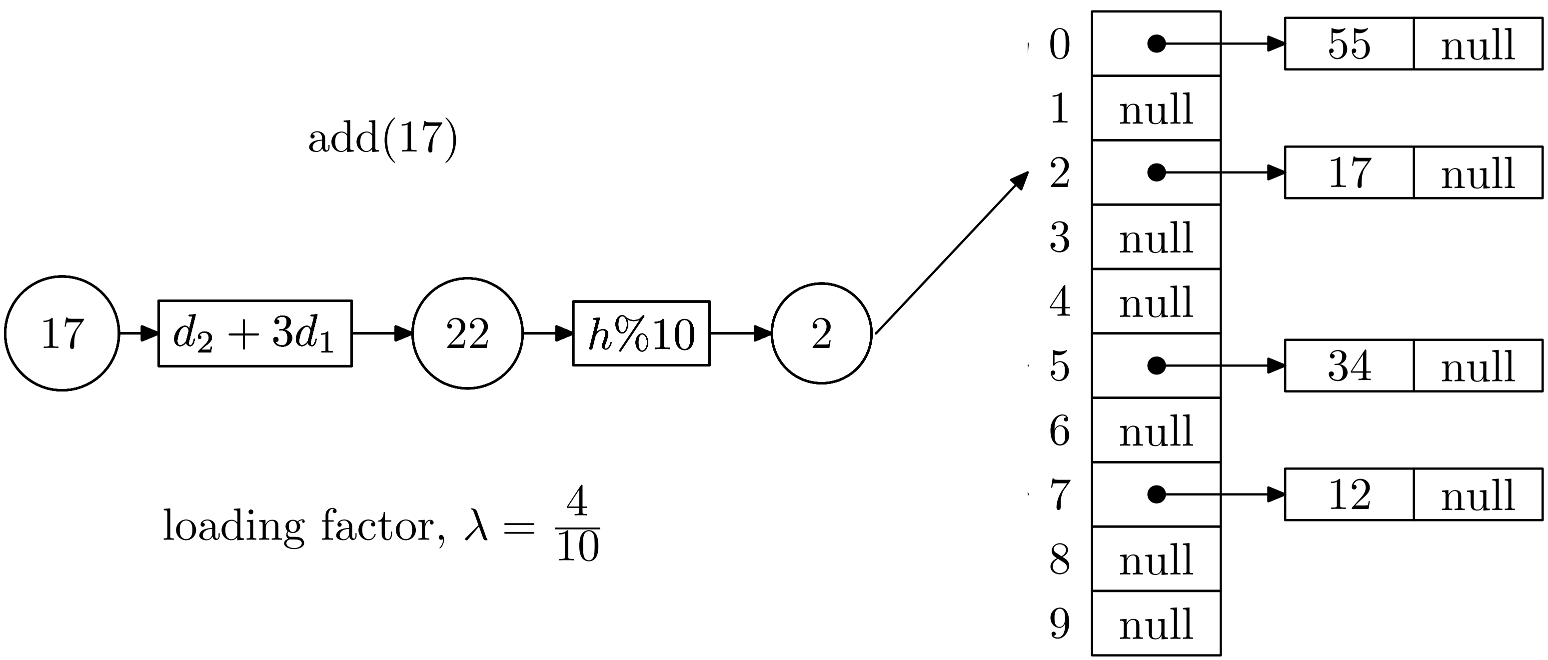
# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



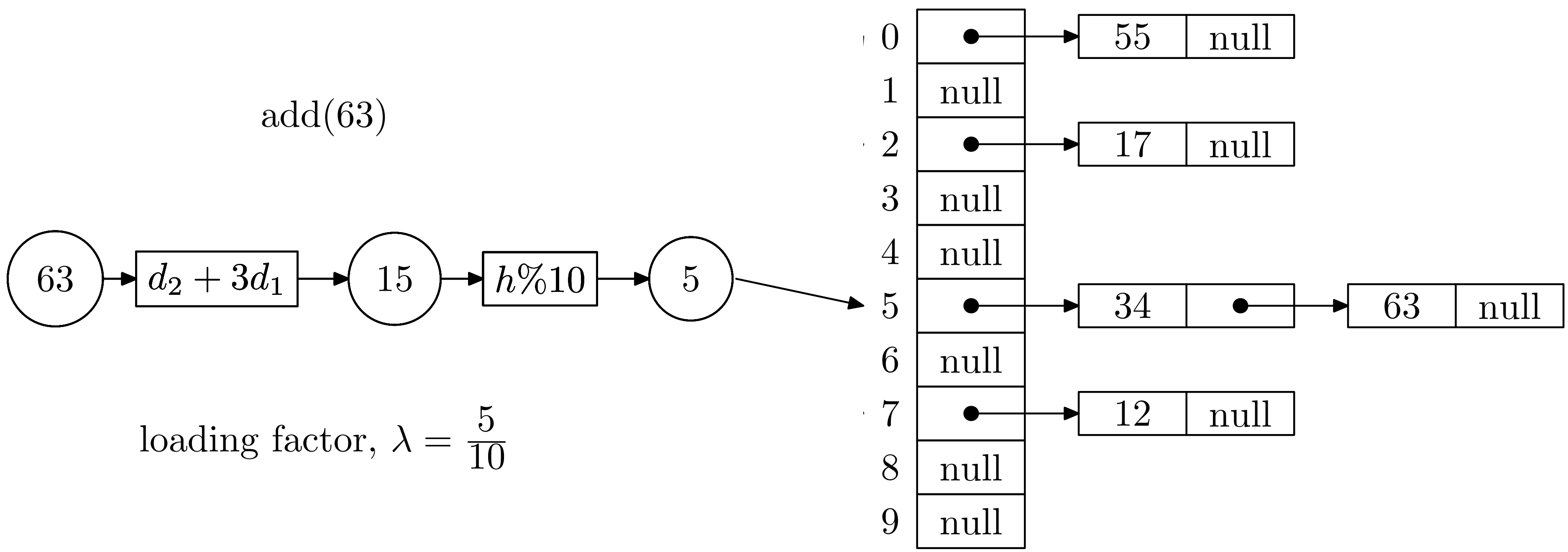
# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



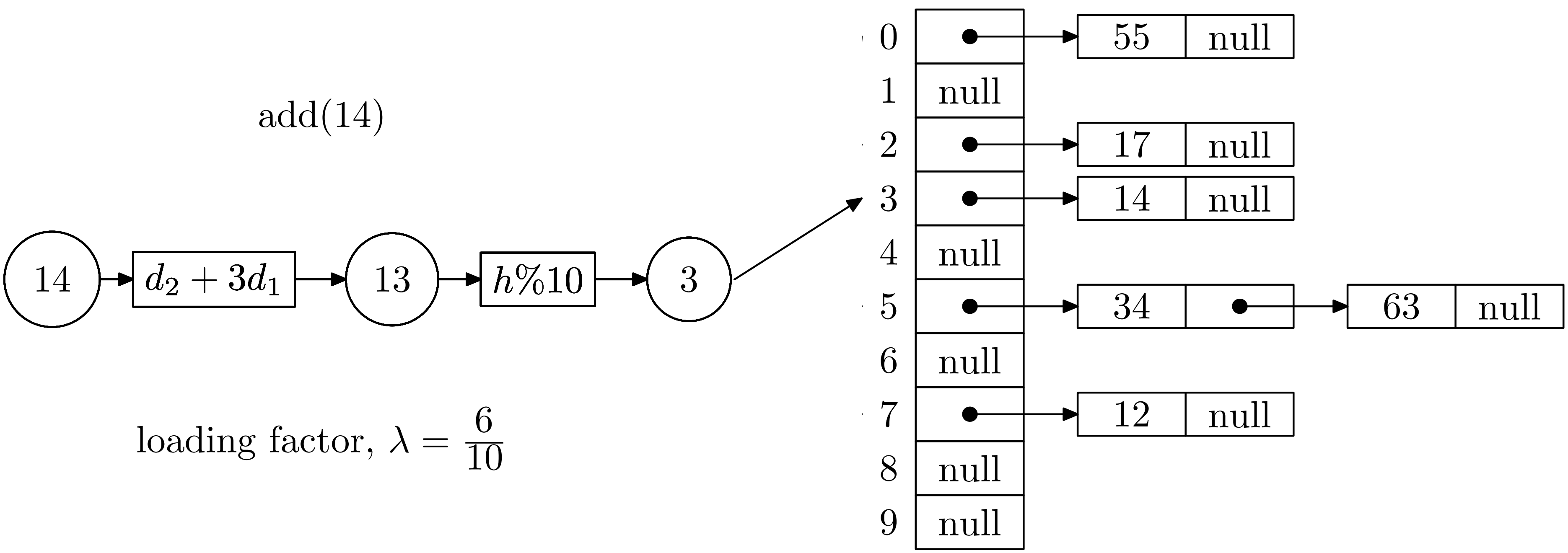
# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



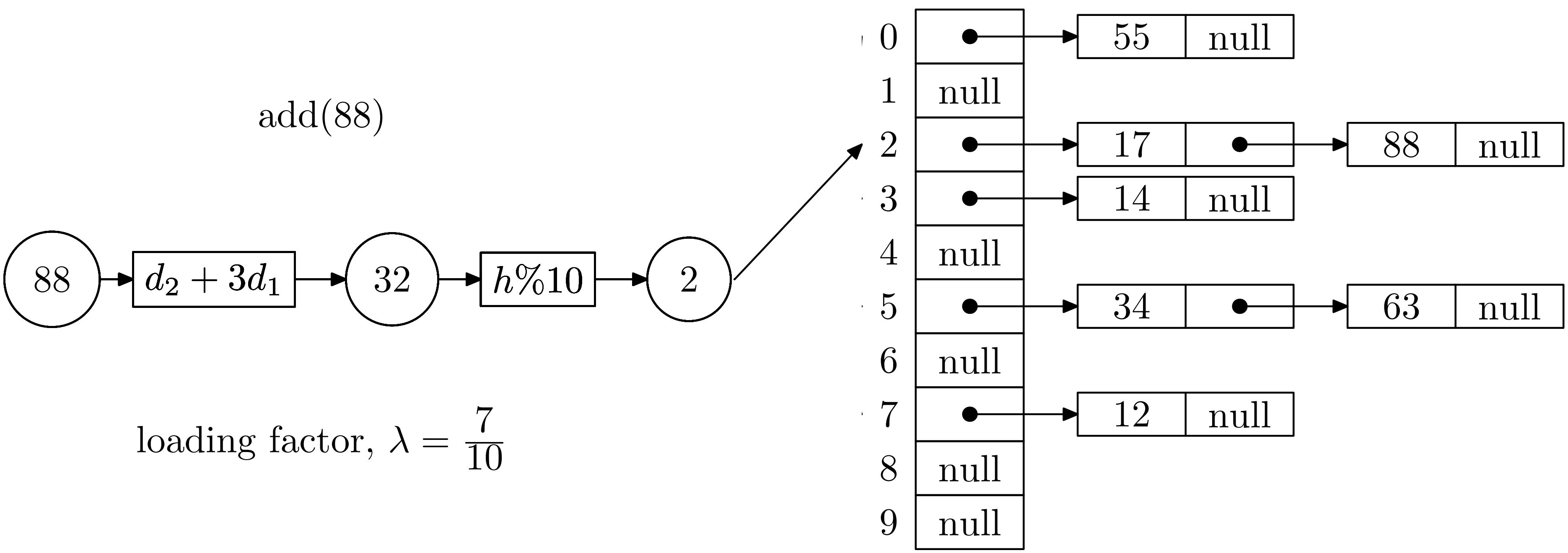
# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



# Separate Chaining

- In separate chaining we build a singly-linked list at each table entry



# Search

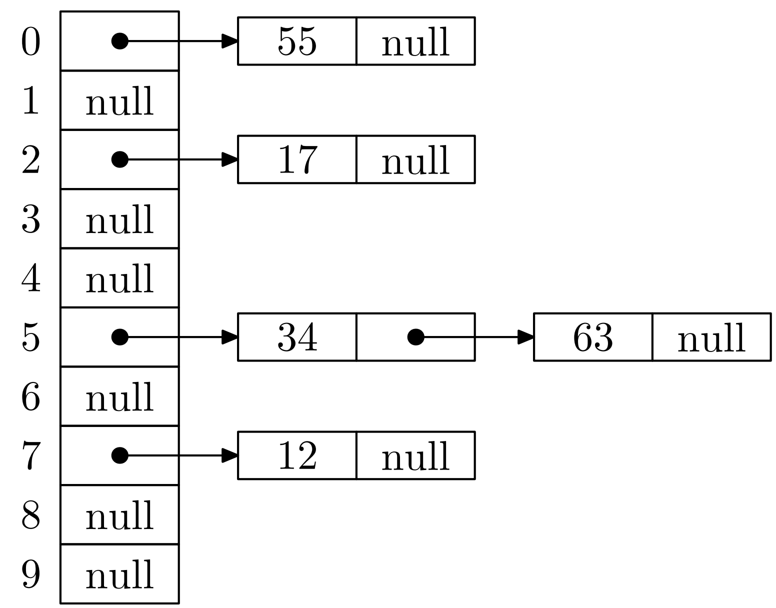
- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list

- The time complexity depends on the objects hashed
  - ⋆ If the objects are evenly dispersed in the table, search (and insertion) is $O(1)$
  - ⋆ If the objects are hashed to the same entry in the hash table then search is $O(n)$

- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance

# Iterating Over a Hash Table

- To iterate over a hash table we
  - ⋆ Iterate through the array
  - ⋆ At each element iterate through the linked list
- The order of the elements appears random
- This becomes more efficient as the table becomes fuller




55, 17, 34, 63, 12

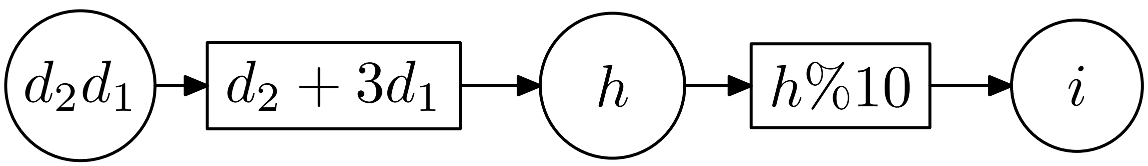
# Outline

1. Why Hash?
2. Separate Chaining
3. **Open Addressing**
   - Quadratic Probing
   - Double Hashing
4. HashSet and HashMap

# Open Addressing

- In open addressing we have a single table of objects (without a linked-list)
- In the case of a collision a new location in the table is found
- The simplest mechanism is known as **linear probing** where we move the entry to the next available location

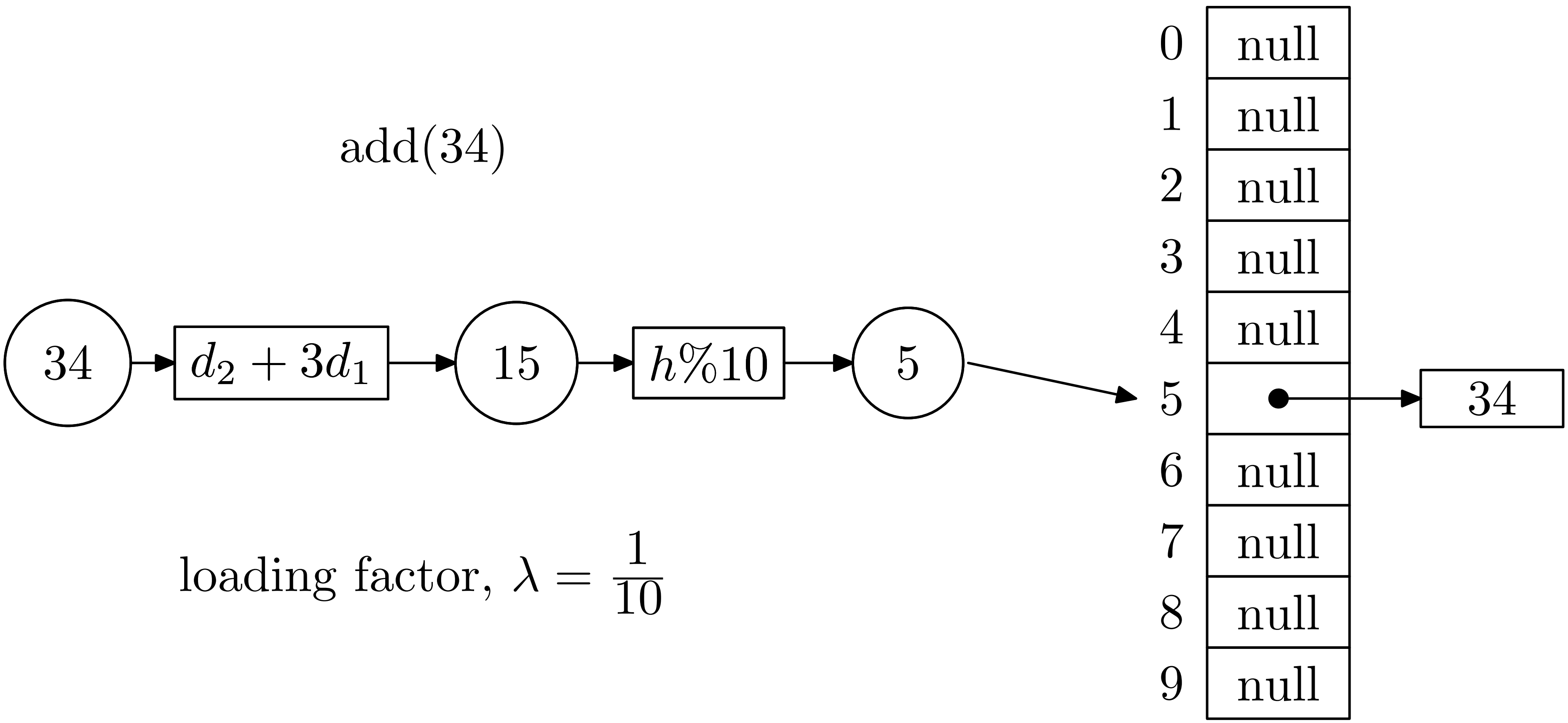
# Linear Probing



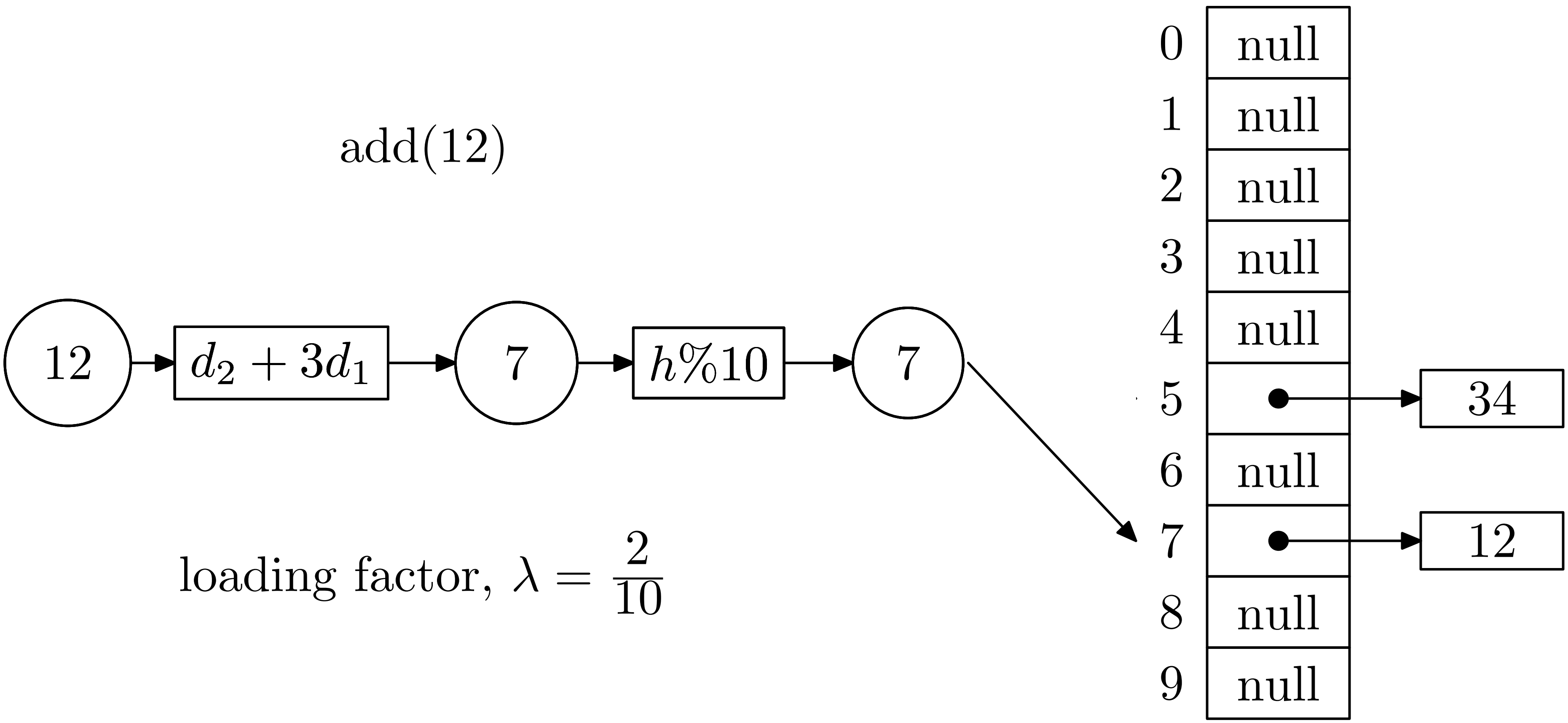

loading factor, $\lambda = \frac{0}{10}$

| | |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| 8 | null |
| 9 | null |

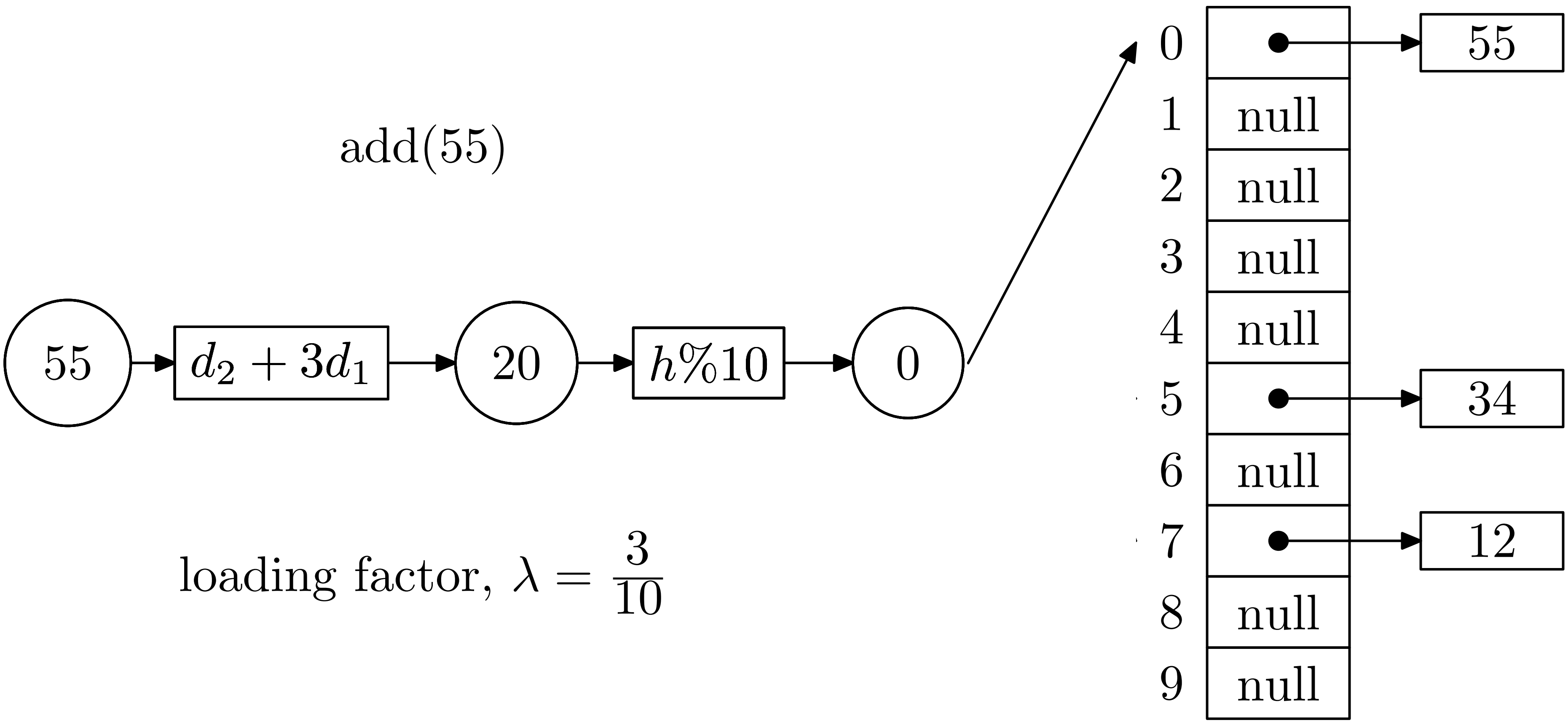
# Linear Probing



# Linear Probing

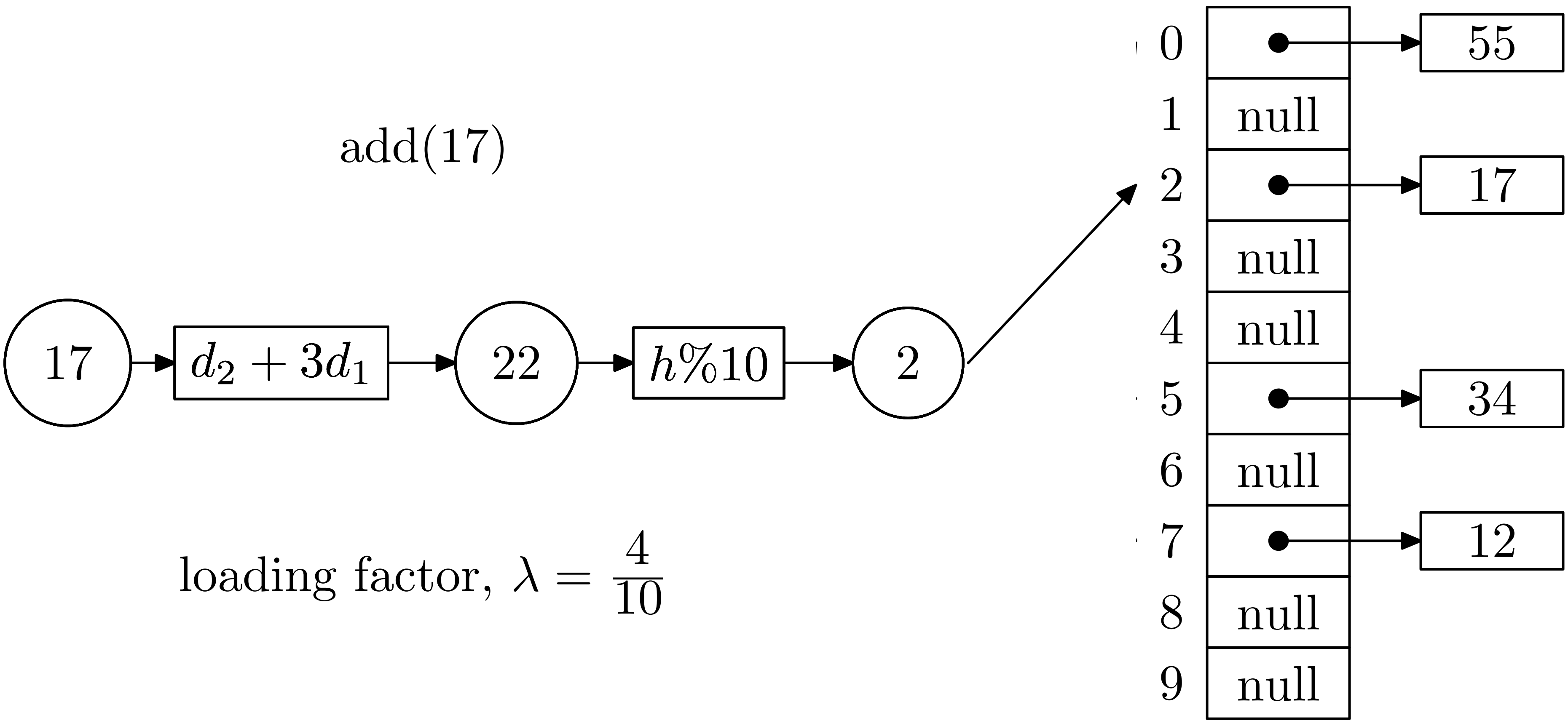


# Linear Probing

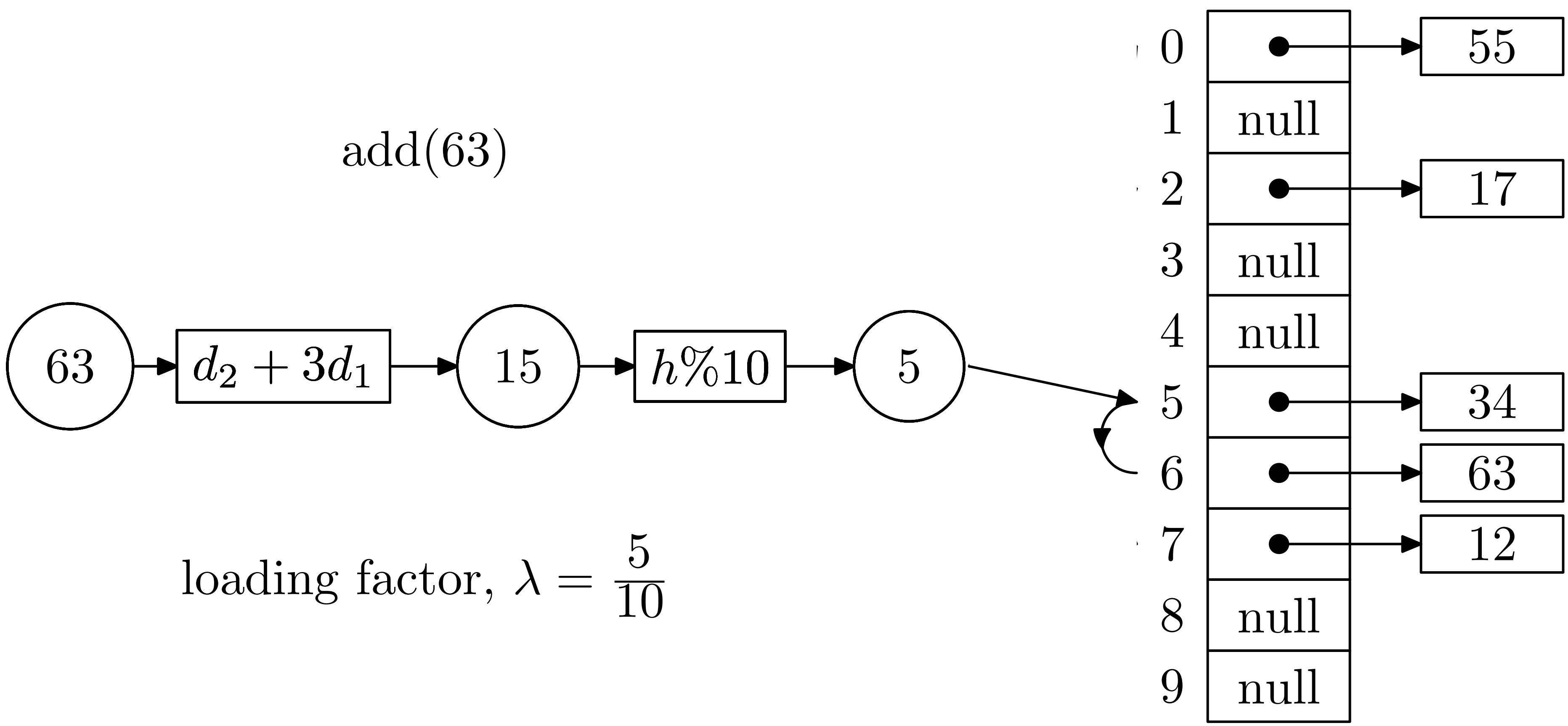



loading factor, $\lambda = \frac{3}{10}$

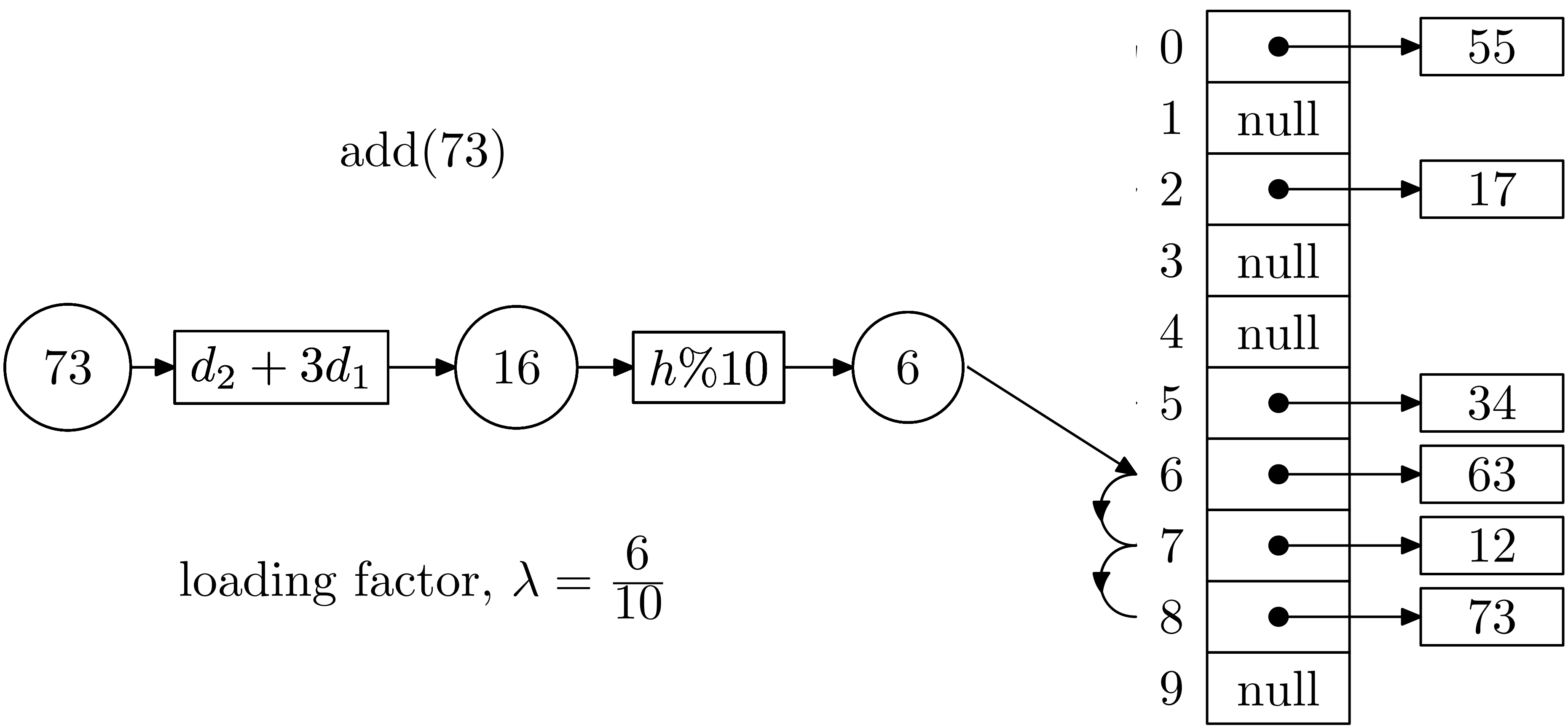
# Linear Probing



# Linear Probing

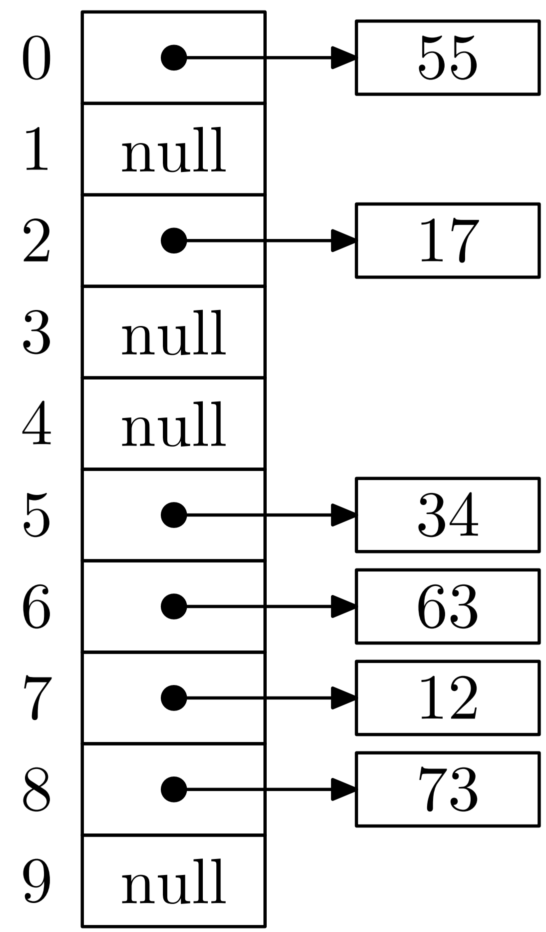


# Linear Probing



# Linear Probing Pile Up

- The entries will tend to pile up or cluster – this is sometimes referred to as **primary clustering**
- Clusters become worse as the number of entries grow
- Clusters will increase the number of probes needed to find an insert location
- The proportion of full entries in the table is known as the **loading factor**



# Reducing Number of Probes

- The number of probes needed for insertion and searching increases with the loading factor, $\lambda$
- With linear probing this is made worse by the tendency to cluster
- E.g. for a loading factor $\lambda = 0.9$ $(1$ in $10$ locations is free)
  - ⋆ without clustering the expected number of probes would be 10
  - ⋆ linear probing typically requires ≈50 probes for insertion
- To avoid clustering two other strategies are commonly used
  - ⋆ Quadratic probing
  - ⋆ Double hashing

# Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$
- That is we takes steps 1, 4, 9, 16, . . .
- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full
- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full
- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

# Double Hashing

- An alternative strategy is known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$
- $h_2(x)$ is a second hash function that depends on the key
- A good choice is $h_2(x) = R - (x\%R)$ where $R$ is a prime smaller than the table size
- It is important that $h_2(x)$ is not a divisor of the table size
  - ⋆ Make sure the table size is prime!

# Problems with Remove

- For all open addressing hash systems removing an entry is a problem

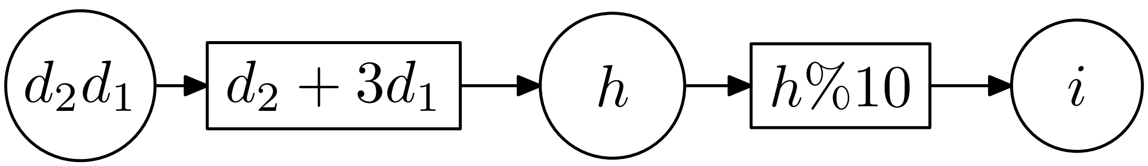
- Remember our strategy to find an input $x$ is

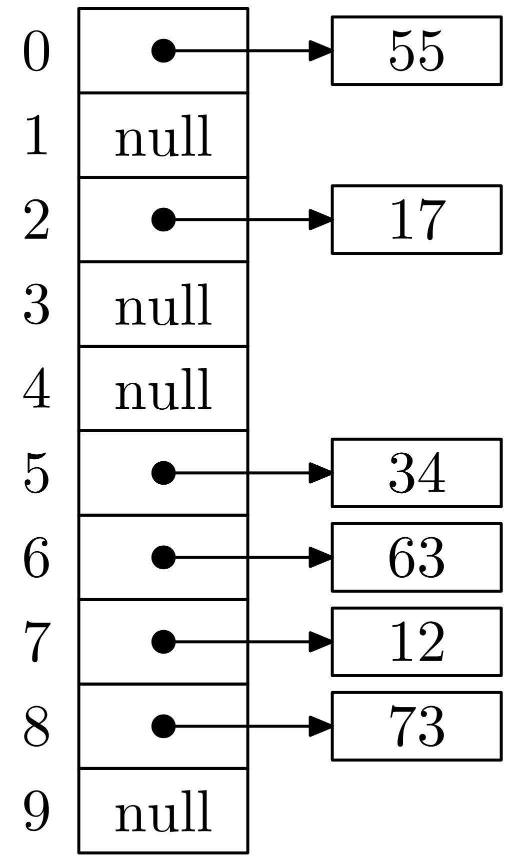
  1. Compute the array index based on the hash code of $x$
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

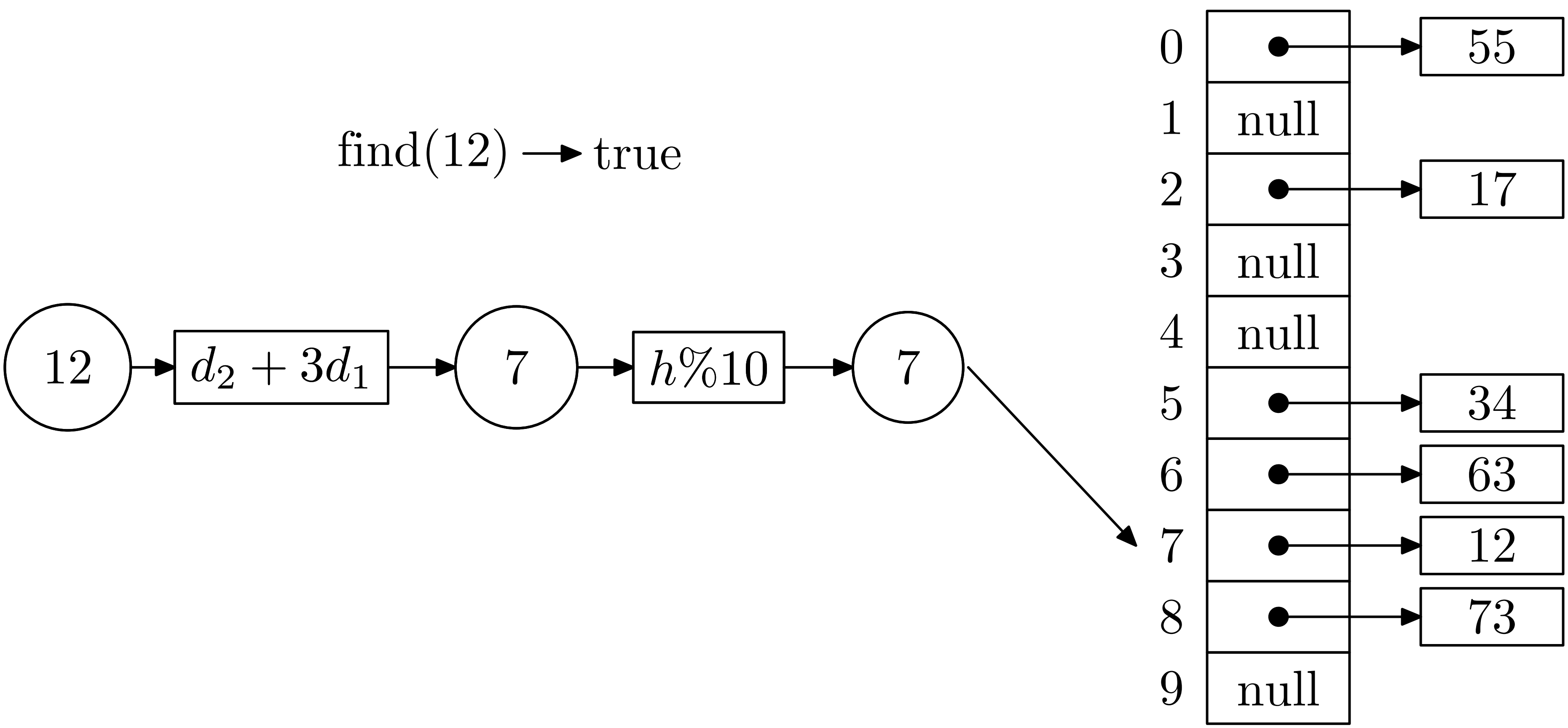
- This can prevent us finding a true entry

# Linear Probing Example

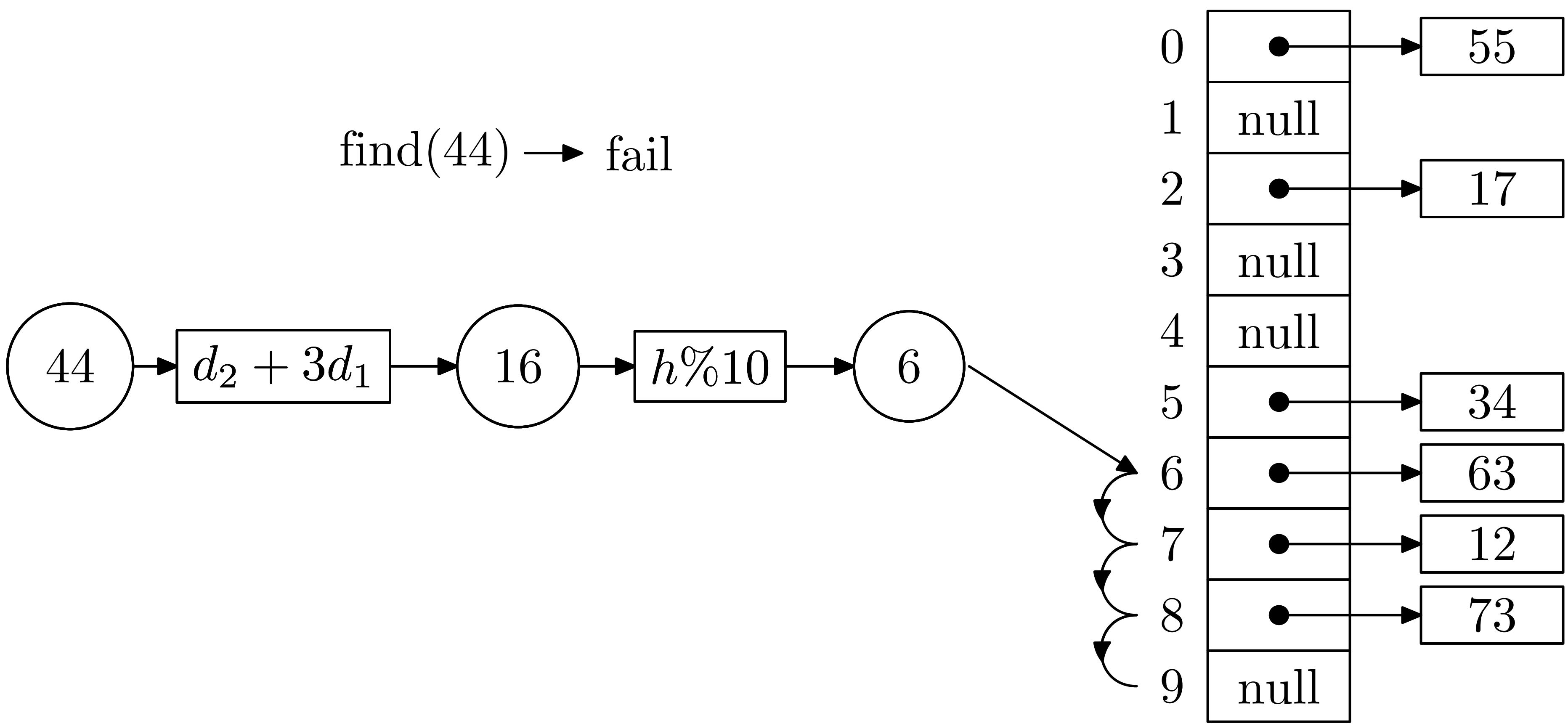


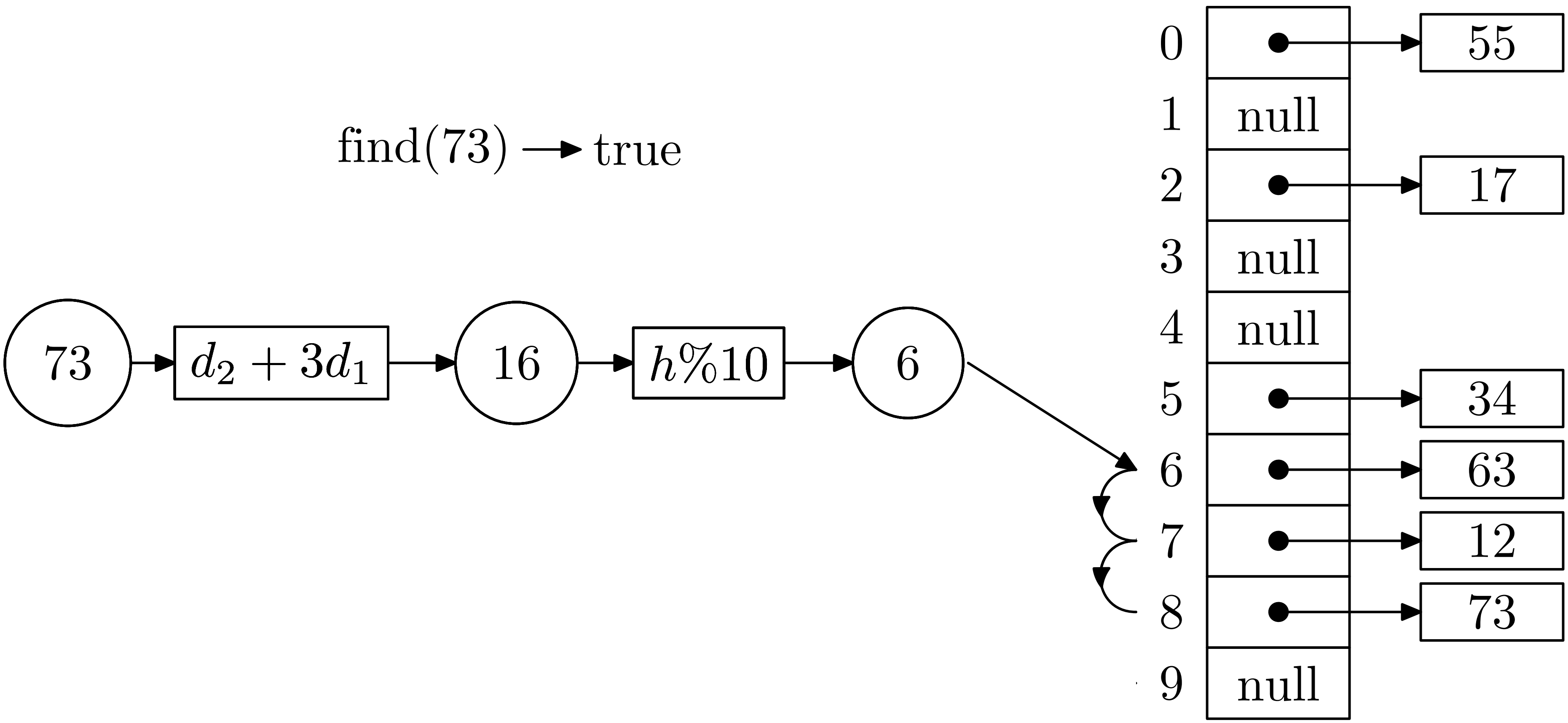
# Linear Probing Example



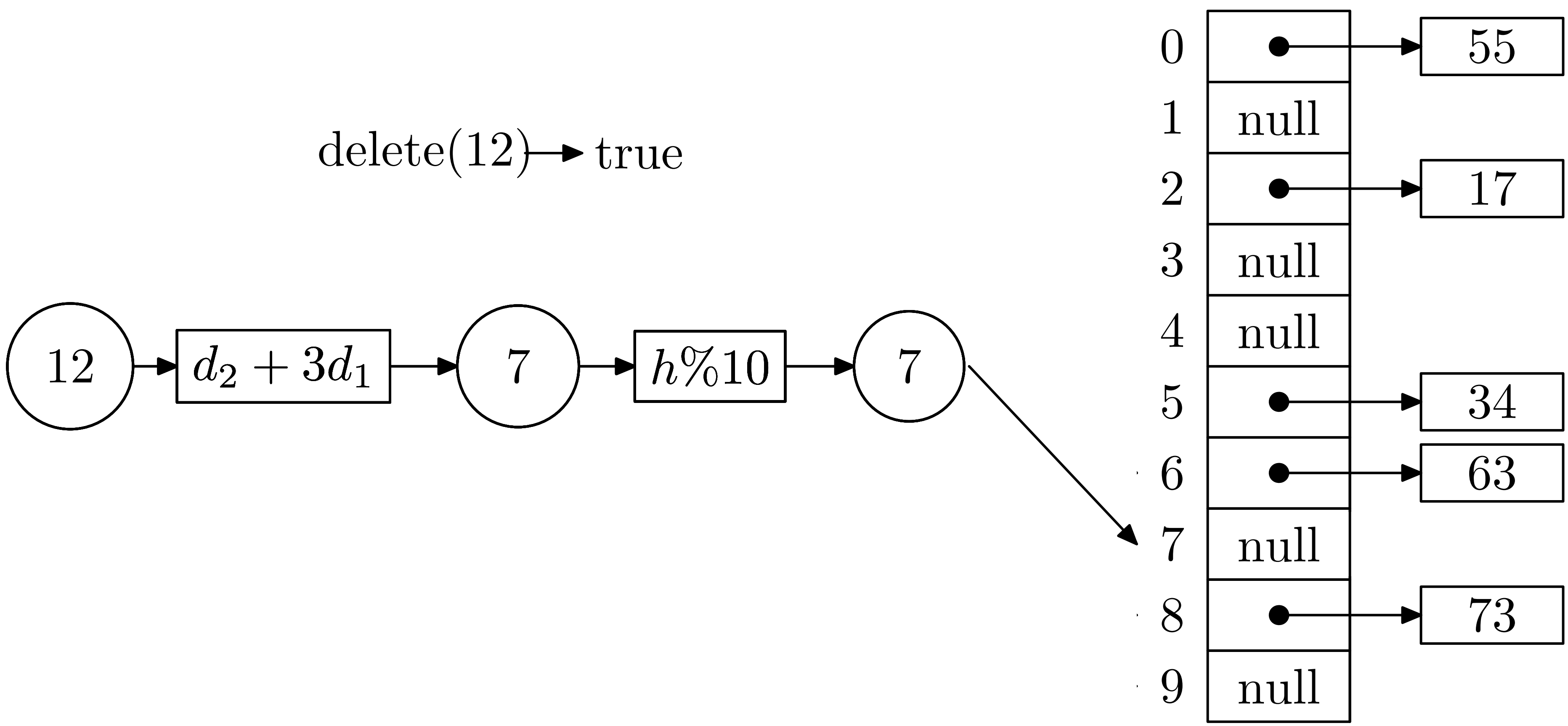
# Linear Probing Example



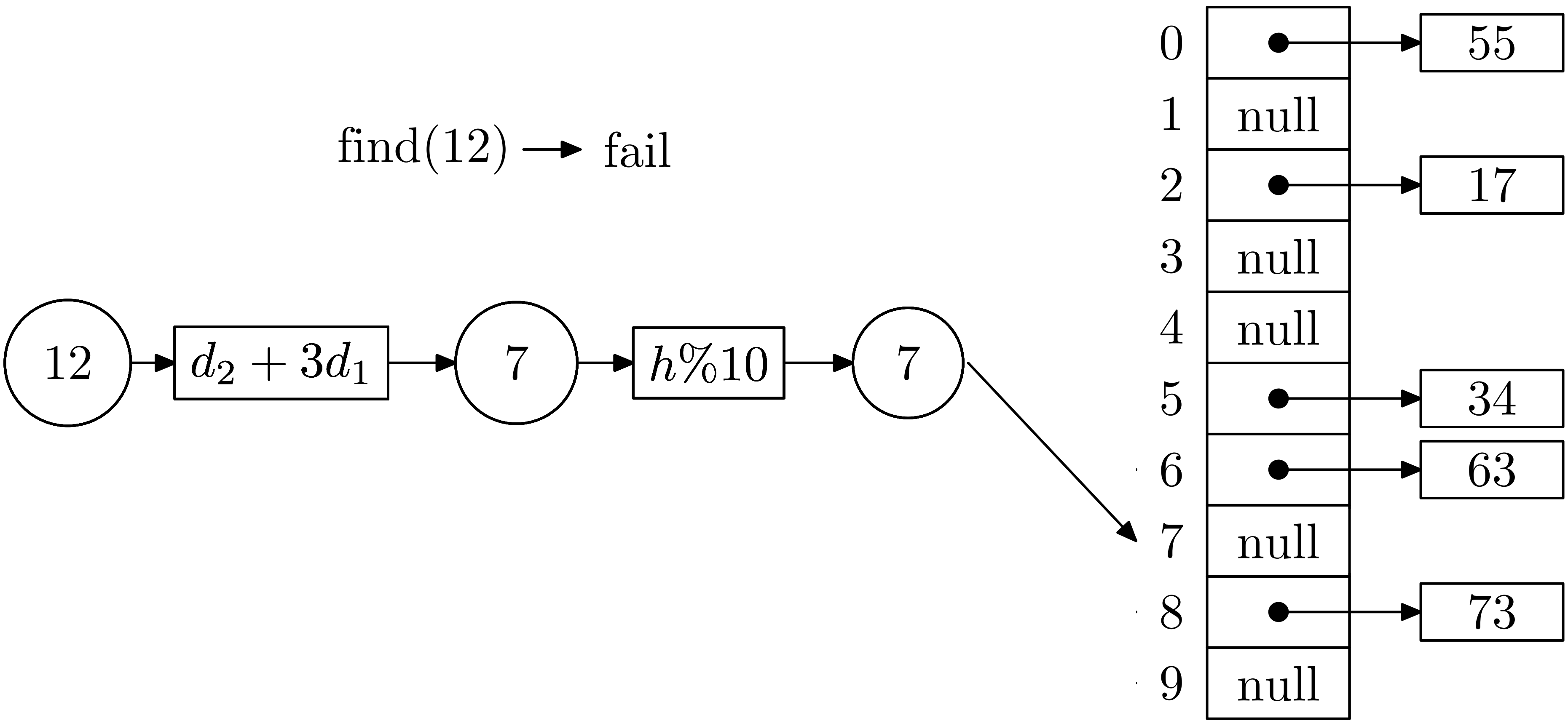
# Linear Probing Example



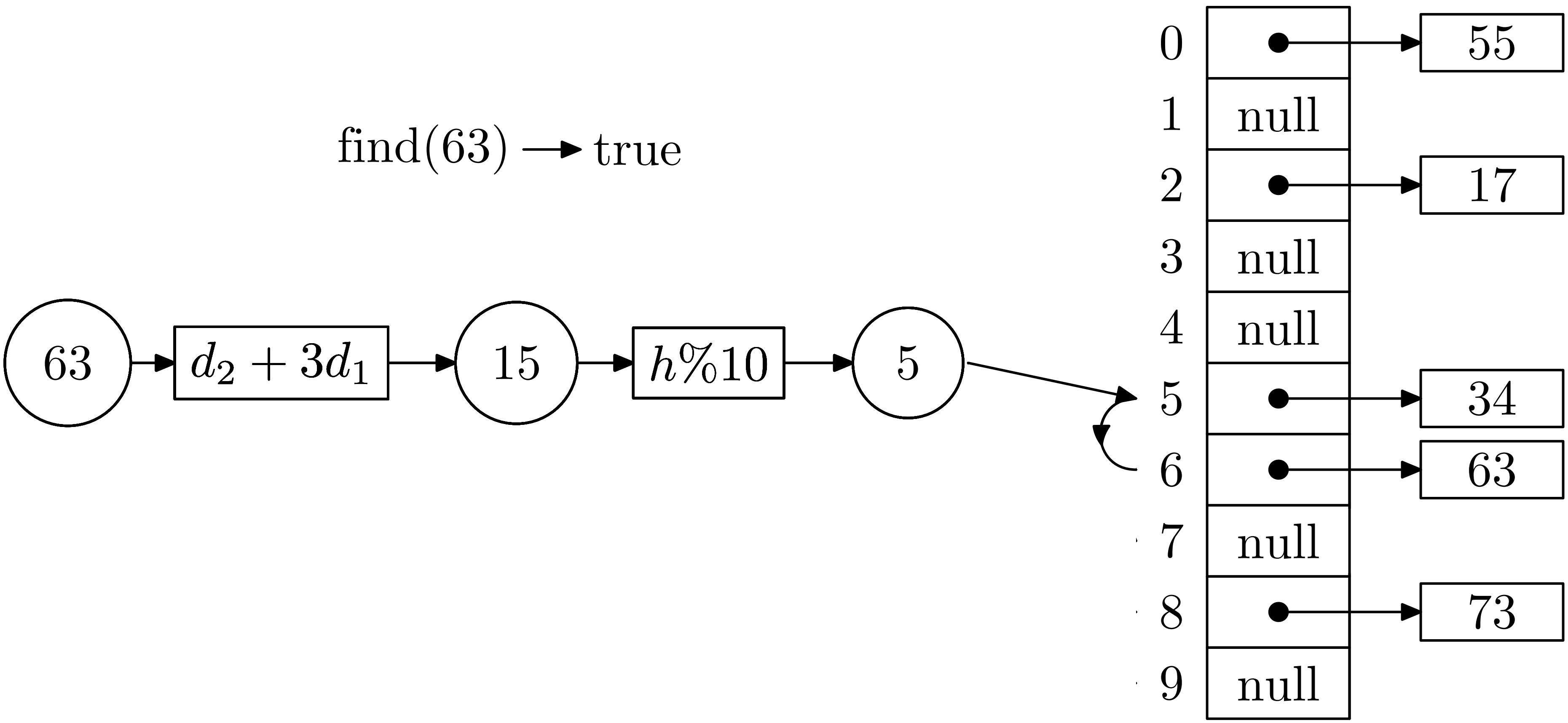
# Linear Probing Example



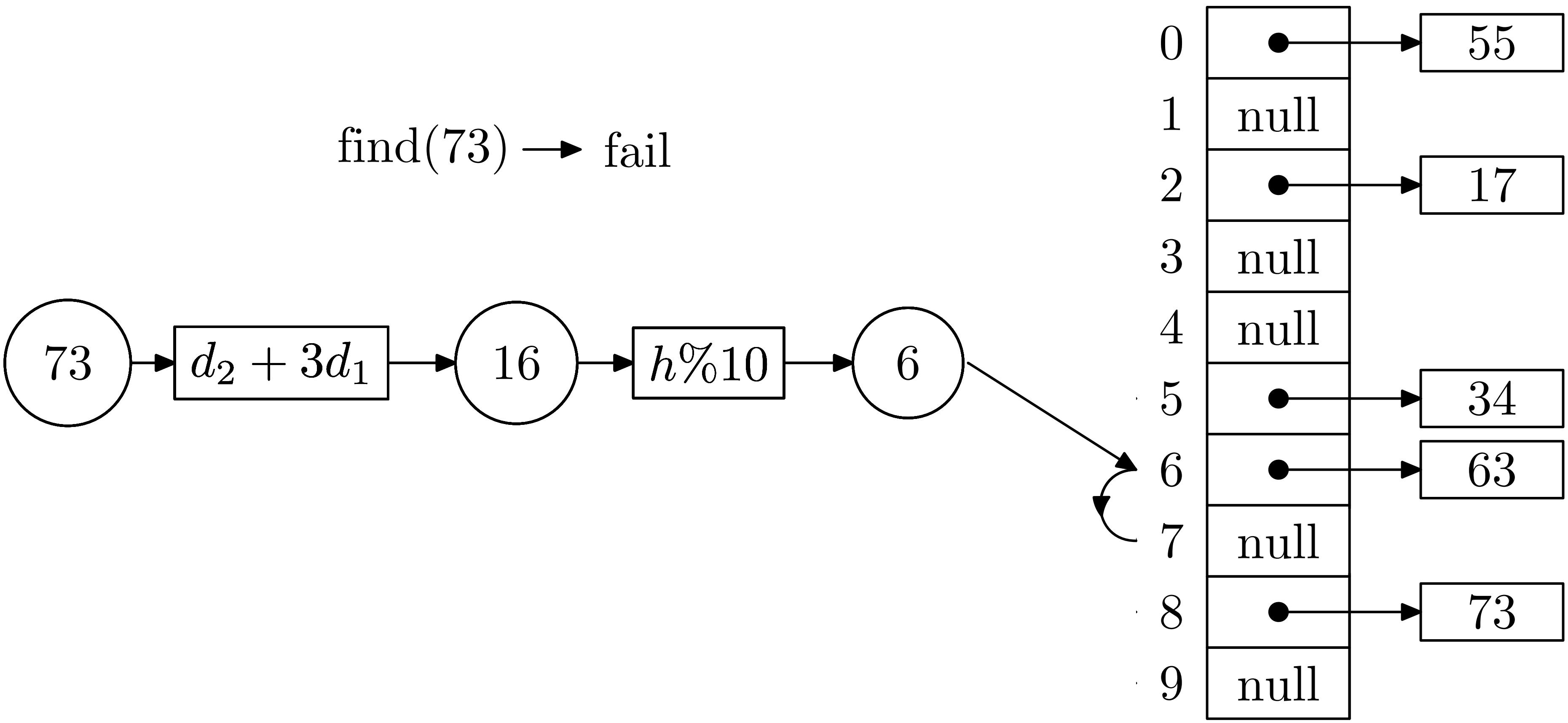
# Linear Probing Example



# Linear Probing Example



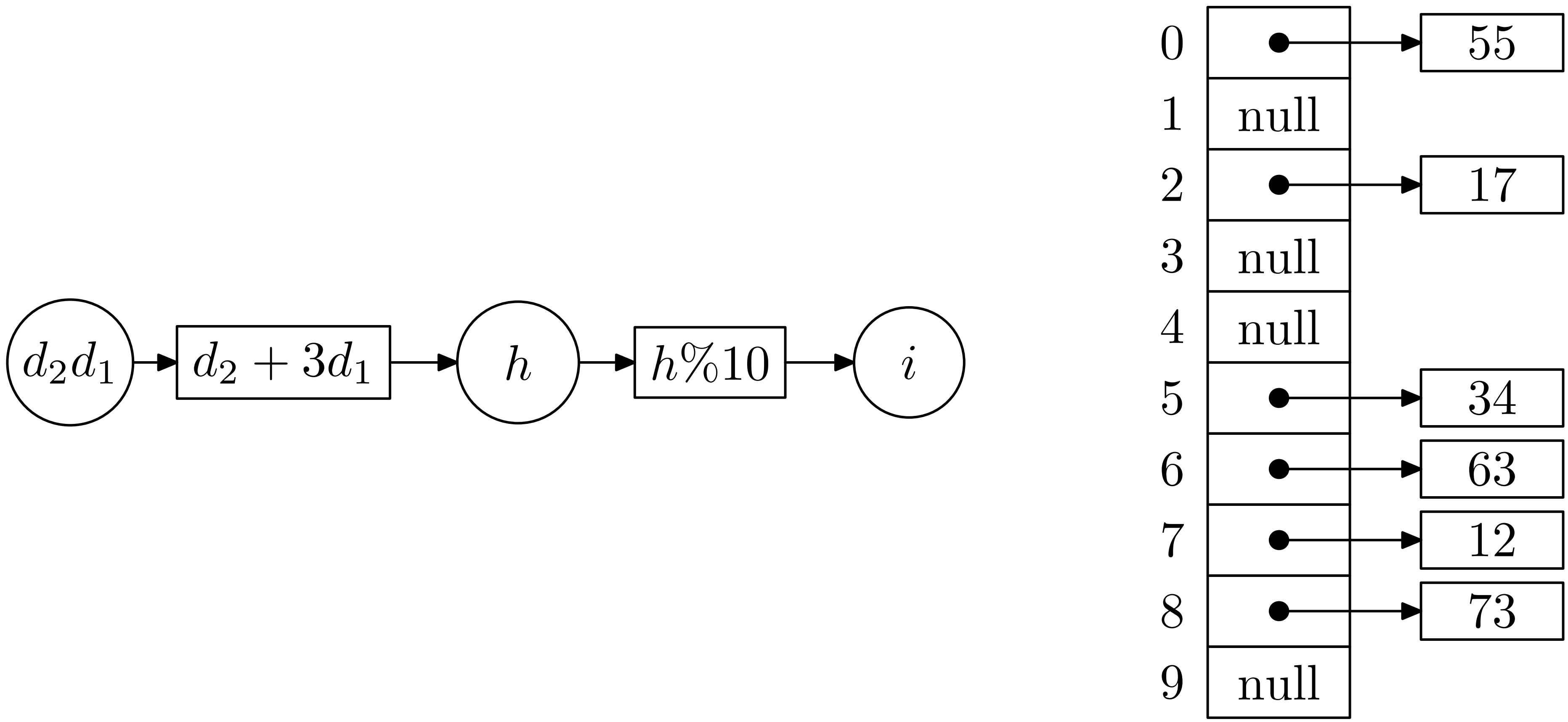
# Linear Probing Example



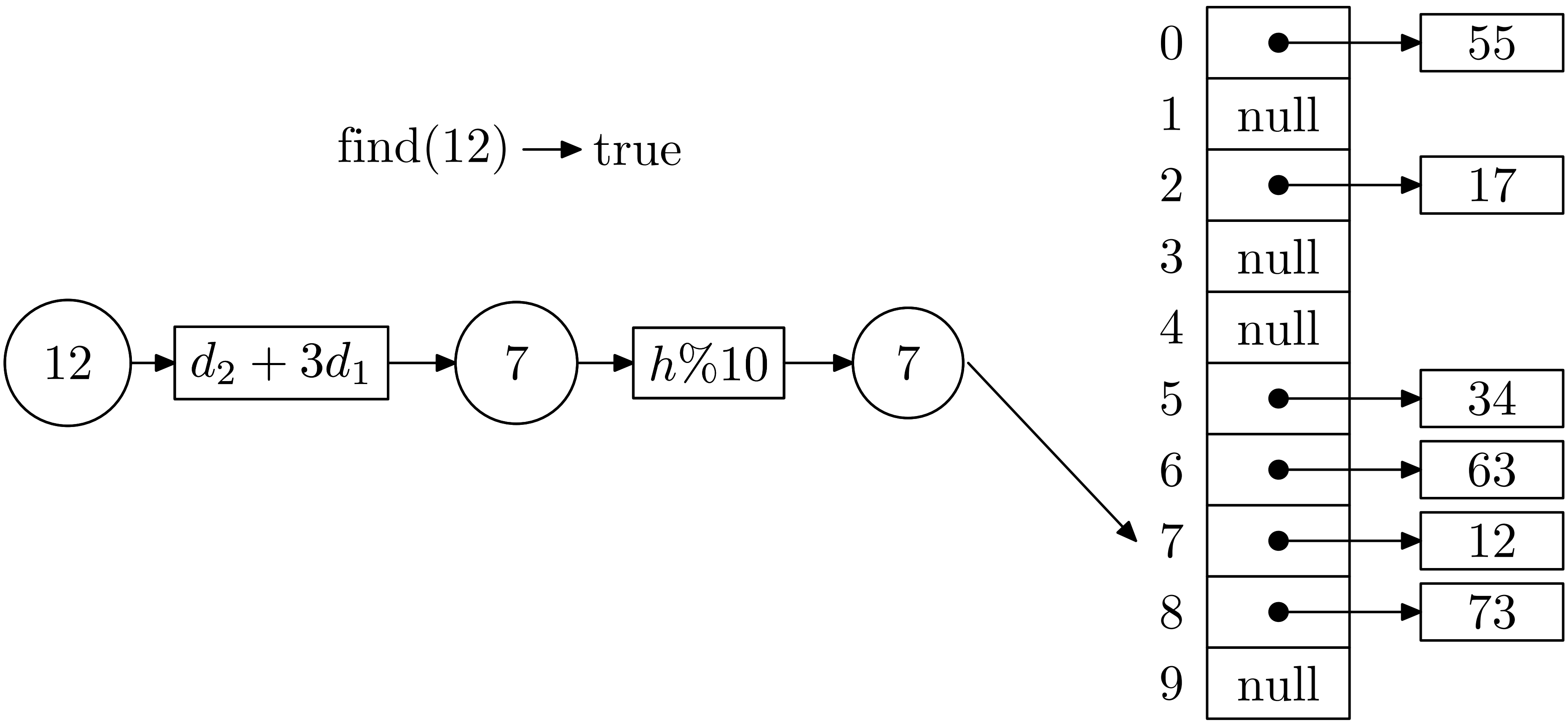
# Lazy Remove

- One easy fix is to mark the deleted table with a special entry
- A find method would consider this entry as full
- An iterator would ignore this entry
- An insert operator could insert a new entry in these special locations

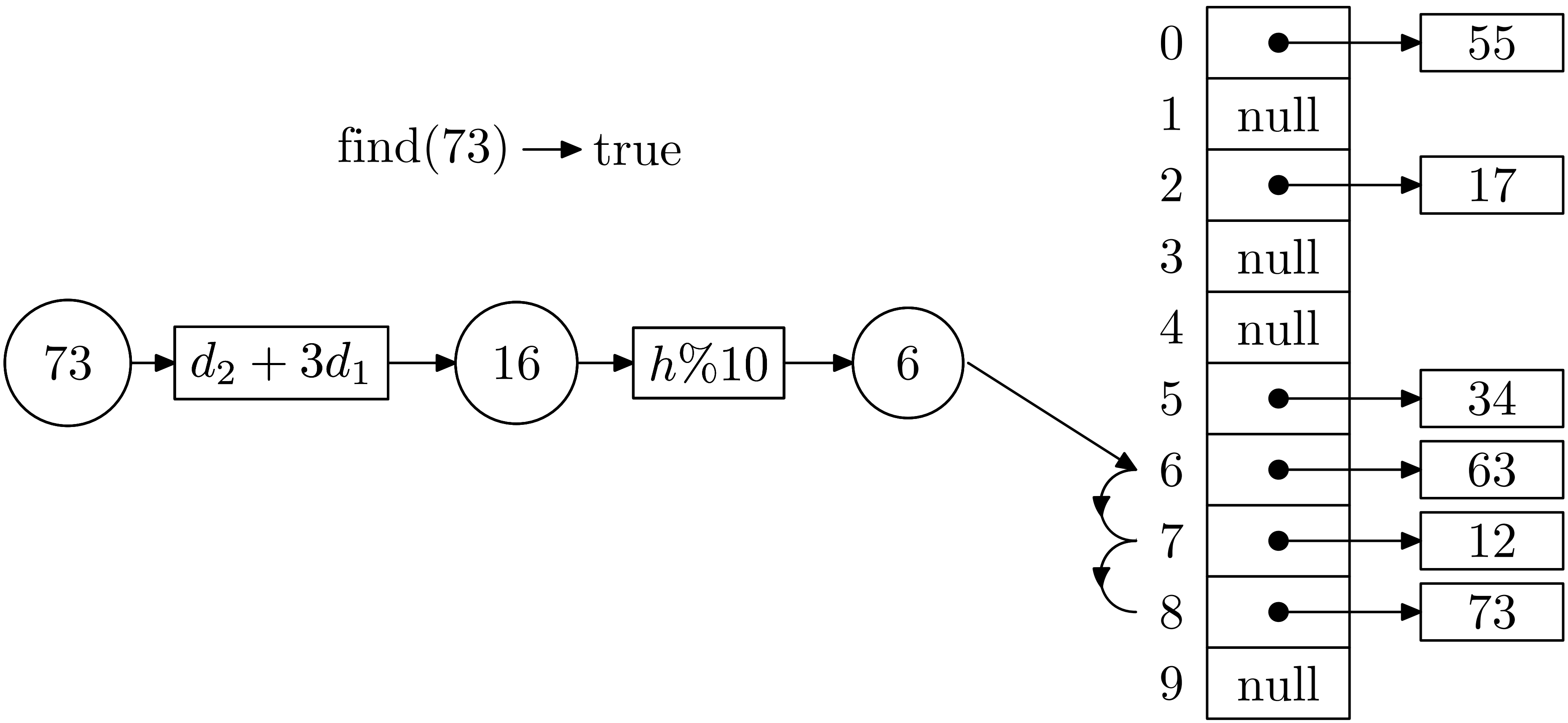
# Lazy Remove in Action



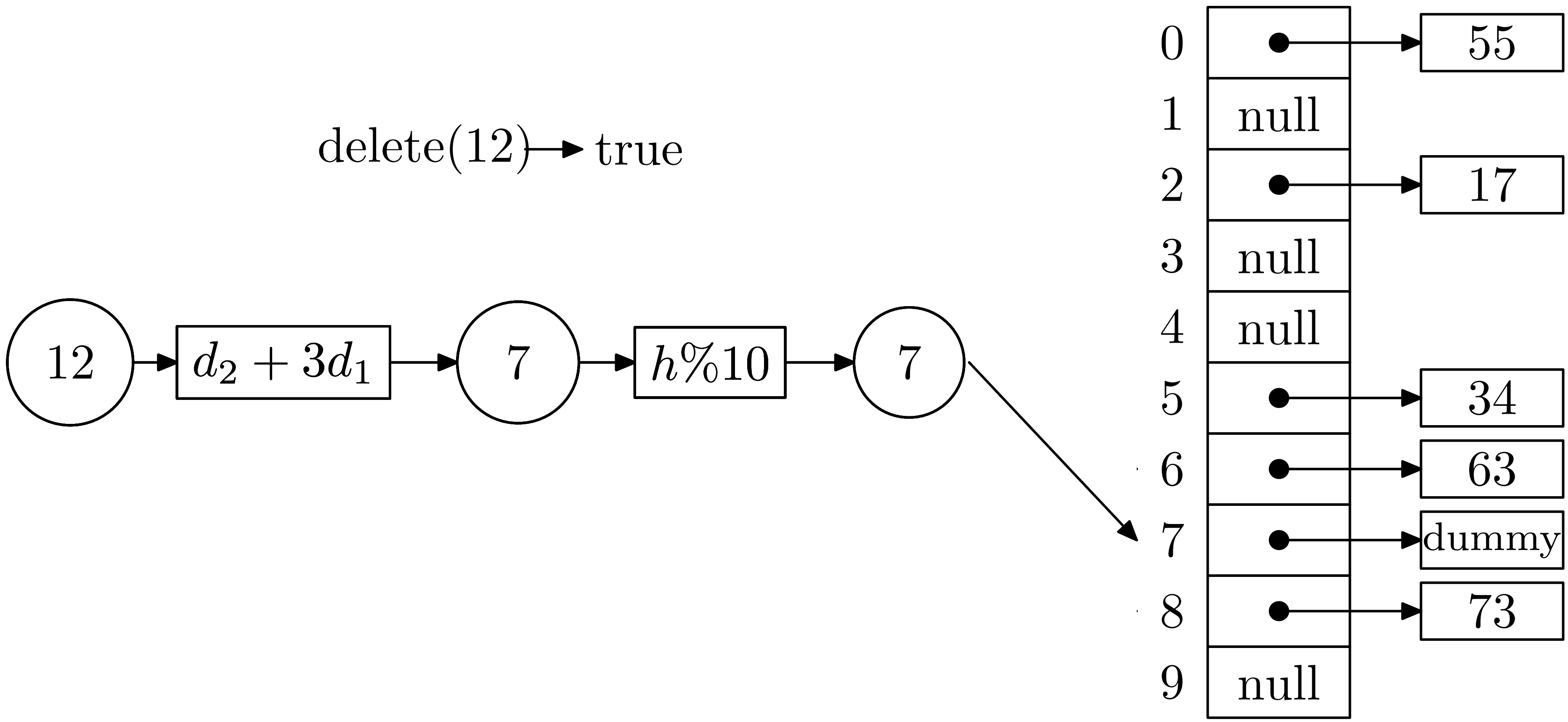
# Lazy Remove in Action



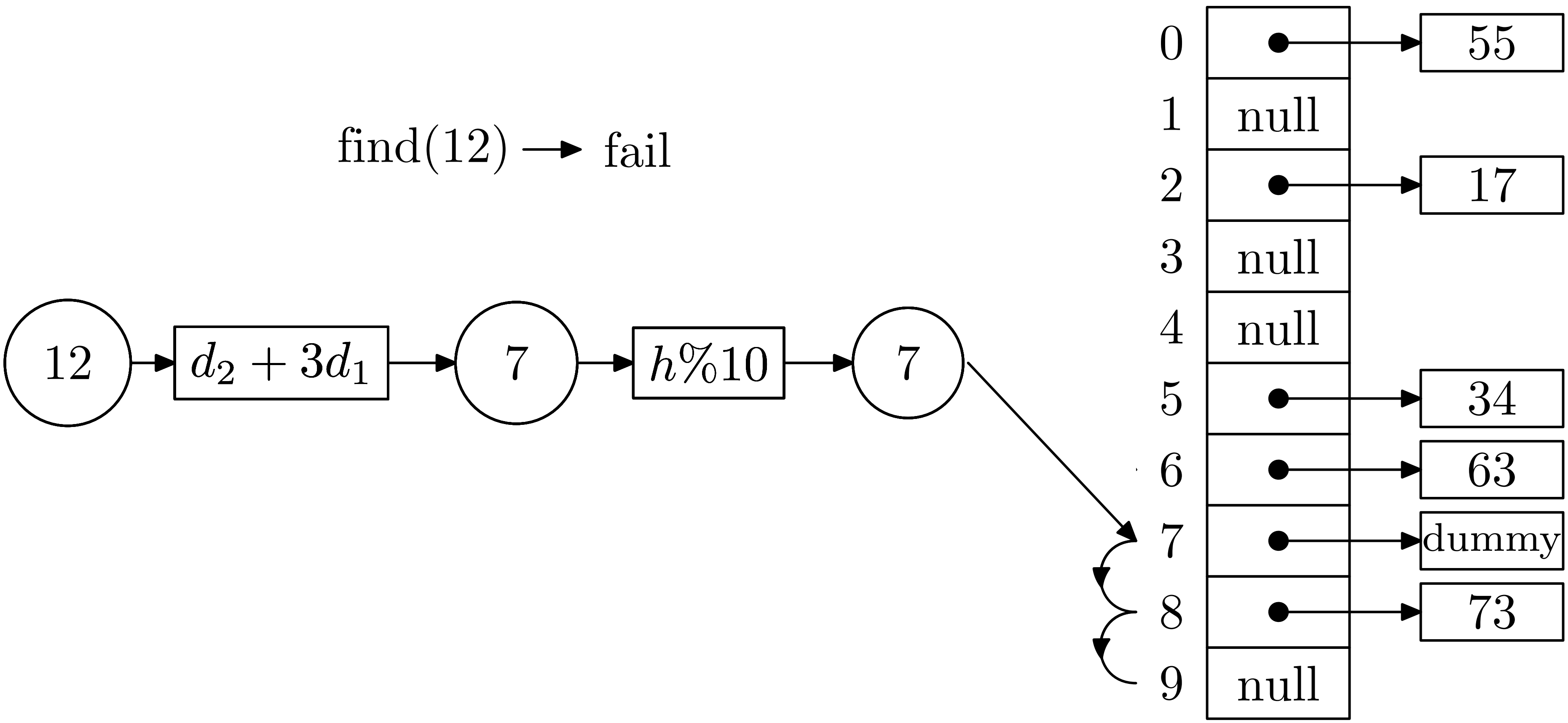
# Lazy Remove in Action



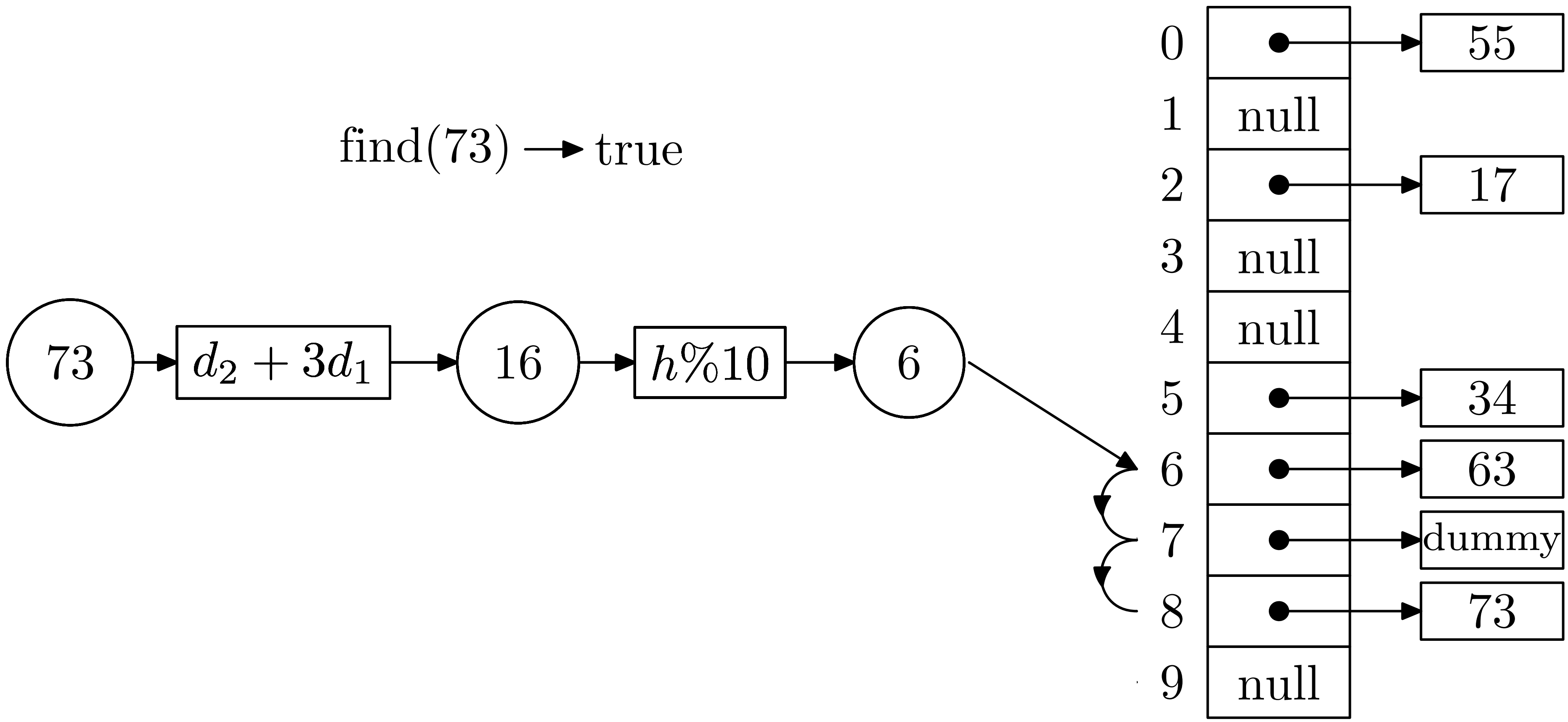
# Lazy Remove in Action



# Lazy Remove in Action



# Lazy Remove in Action



# Outline

1. Why Hash?
2. Separate Chaining
3. Open Addressing
   - Quadratic Probing
   - Double Hashing
4. **HashSet and HashMap**

# What Strategy to Use?

- Most libraries including the Java Collection class use separate chaining
- This has the advantage that performance does not degrade badly as the number of entries increase
- This reduces the need to resize the hash table

# HashSets and HashMaps

- Java provides a `HashSet` and a `HashMap` collection which implements the same `Set` and `Map` interfaces as `TreeSet` and `TreeMap`
- It's performance is asymptotically superior to `TreeSet`, $O(1)$ rather than $O(\log(n))$
- Hash functions can take time to compute, so `HashSet`s might not be faster than `TreeSet`s
- One major difference is that the iterator for `TreeSet`s returns the elements in order, whereas the `HashSet` iterator doesn't!

# Applications

- Hash tables are used everywhere
  - E.g. most databases use hash tables to speed up search
  - In many document applications hash tables are being generated in the background
- Content addressability is ubiquitous to many applications where hash tables are used as standard

# Lessons

- Hash tables are one of the most useful data structures you have available

- They aren't particularly difficult to understand, but you need to know about

  ⋆ hashing functions
  ⋆ collision strategies
  ⋆ performance (i.e. when they work)