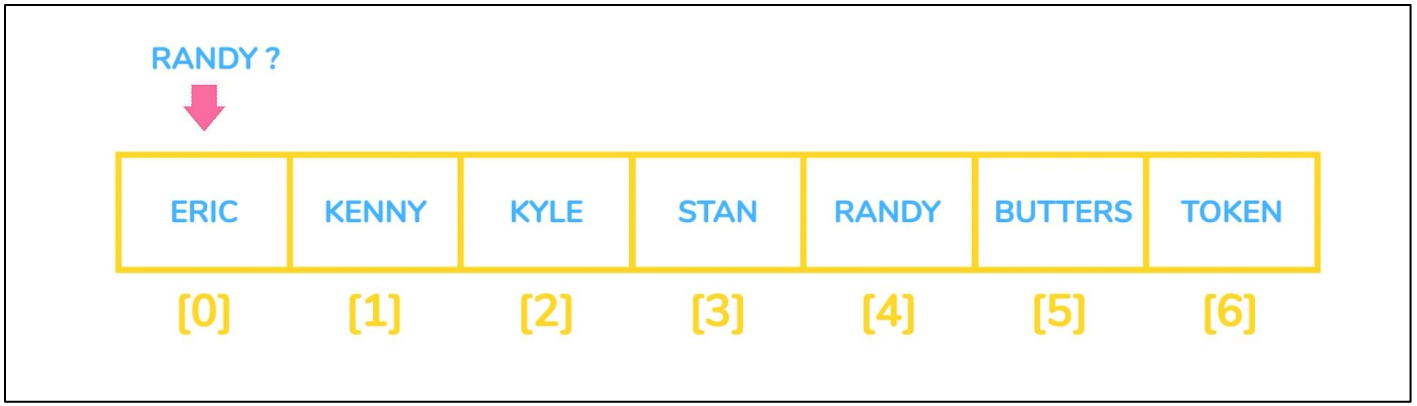
Unit I - Hashing

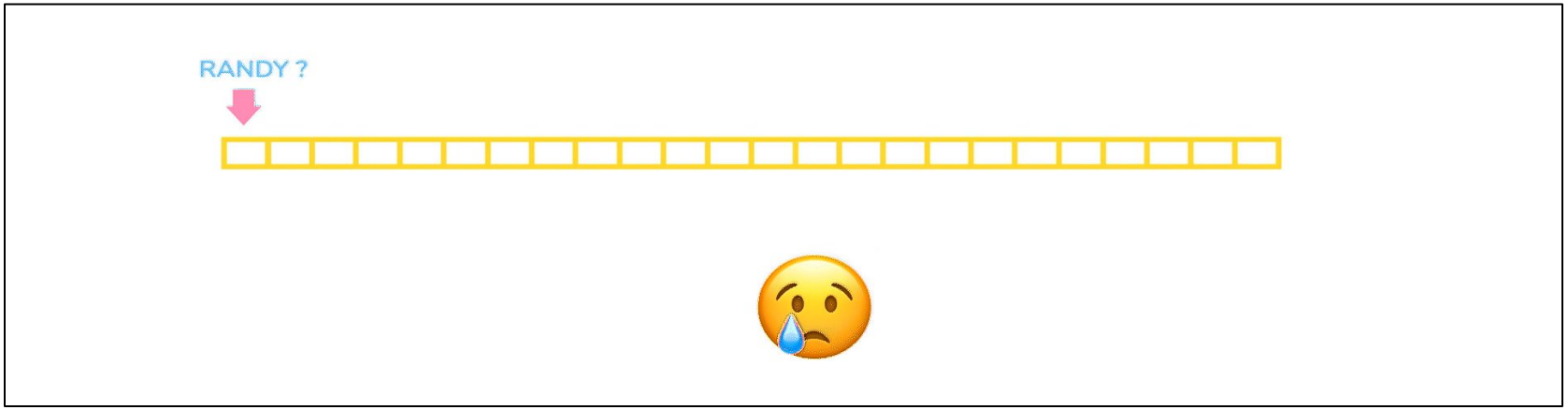
INTRODUCTION

- There is an array, and we've got a bunch of names stored in it. Let's say you want to search for the key "Randy".



INTRODUCTION

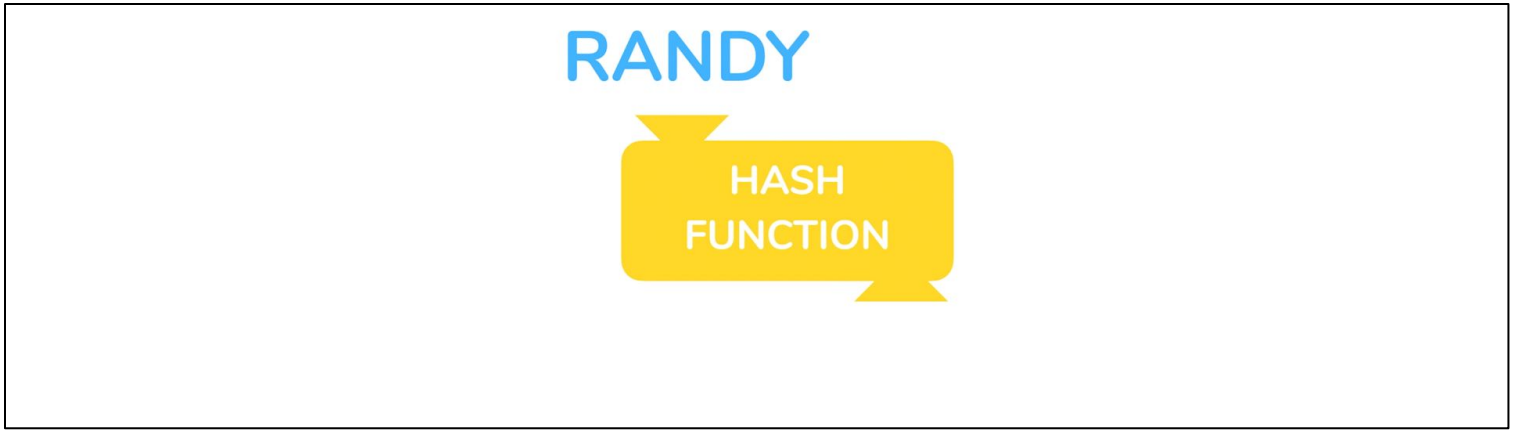
- The array above is pretty simple, but in reality, arrays are are larger than this, and you would be like…



INTRODUCTION

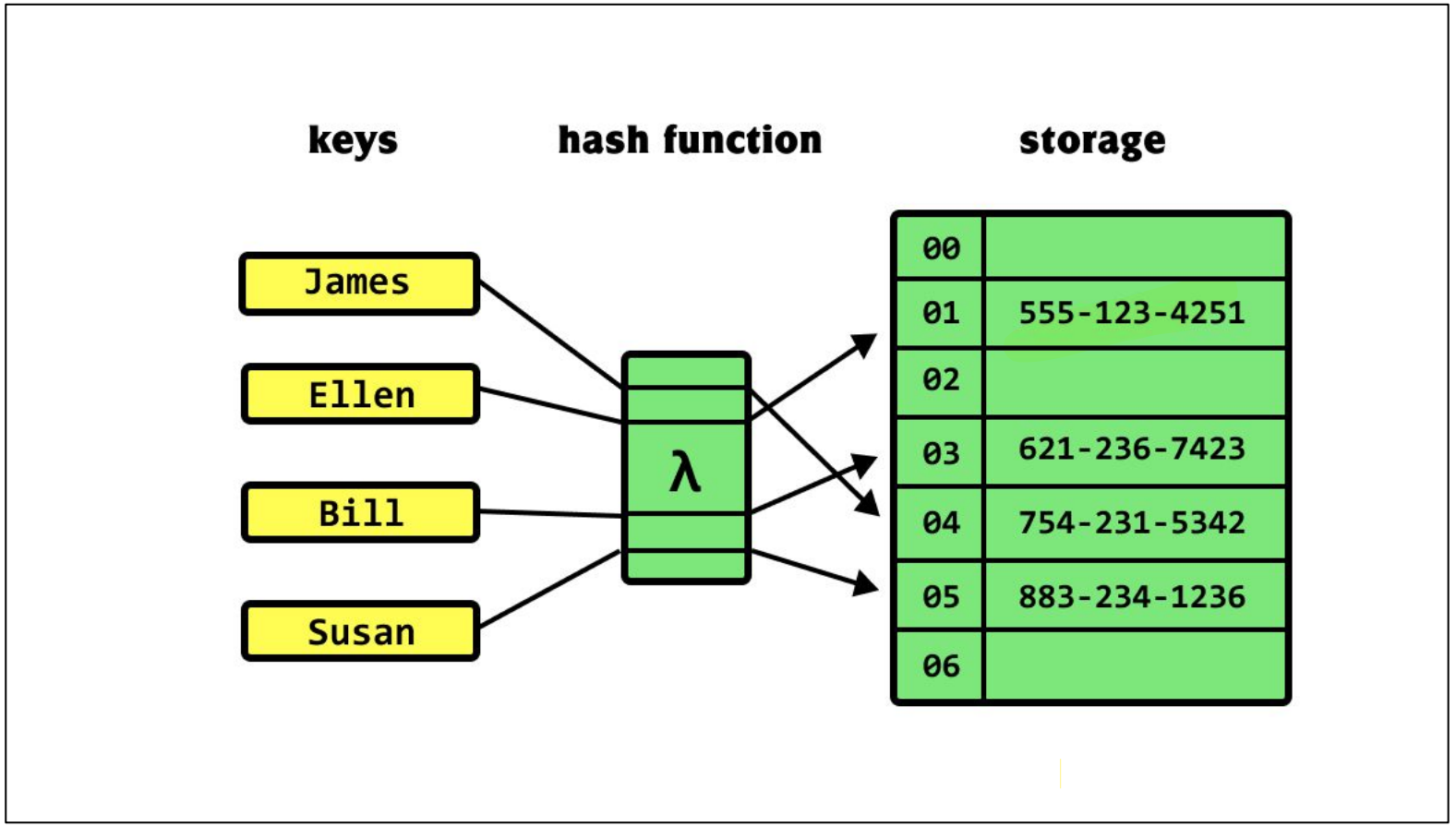
- **Hashing** is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the **hash function.**
- **A hash table** is an array-ba sed structure used to store <key, information> pairs.
- Hashing is an approach in which time required to search an element doesn't depend on the total number of elements.
- Using hashing data structure, a given element is searched with **constant time complexity**.
- Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

- **Definition** - Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.
- Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.



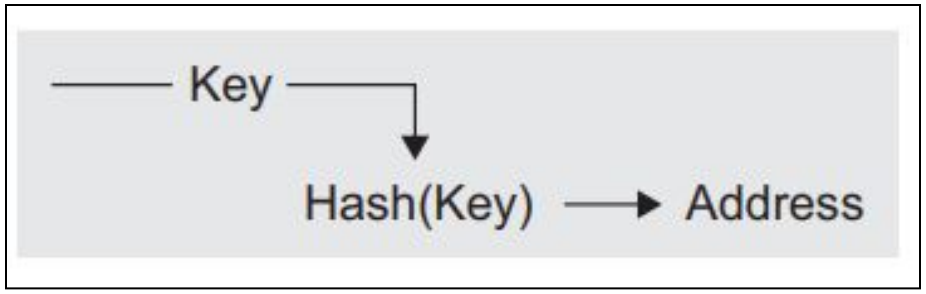
Hash Table

- A data structure that has data stored in this way is called a hash table.



Hash Functions

- Hash functions transform a key into an address.
- Hashing is a technique used for storing and retrieving information associated with it that makes use of the individual characters or digits in the key itself.

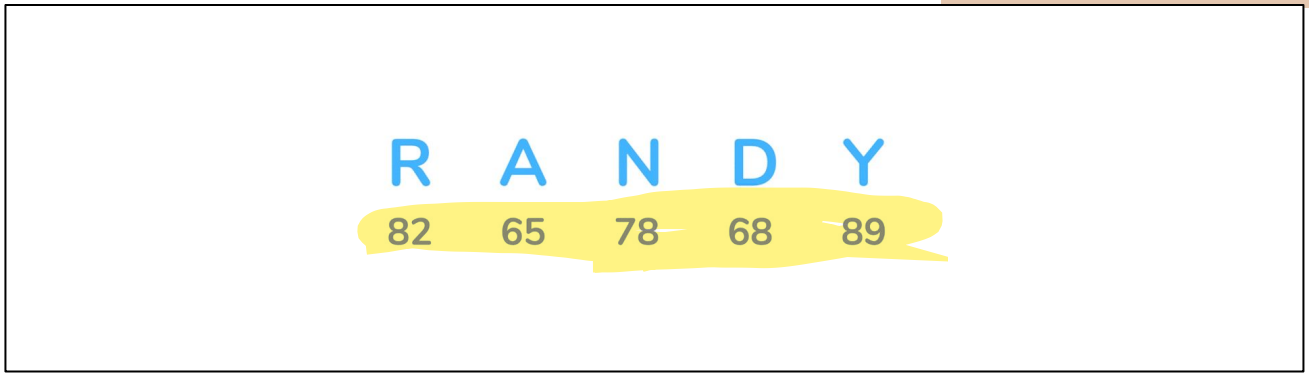


- The resulting address is used as the basis for storing and retrieving records and this address is called the home address of the record.

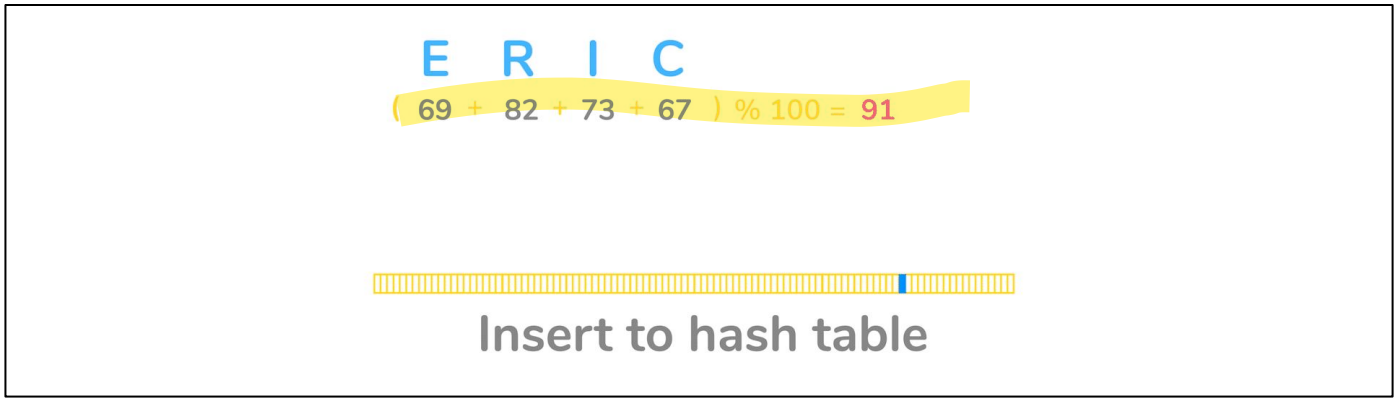
- For an array to store a record in a hash table, the hash function is applied to the key of the record being stored, returning an index within the range of the hash table.
- The item is then stored in the table at that index position.
- To retrieve an item from a hash table, the same scheme that was used to store the record is followed.
- Hashing is similar to indexing as it involves associating a key with a relative record address.

KEY TERMS AND ISSUES in Hashing

- **Hash table -** Hash table is an array [0 to Max – 1] of size Max.
- **Hash function –**
  - Hash function is one that maps a key in the range [0 to Max – 1], the result of which is used as an index (or address) in the hash table for storing and retrieving records.
  - One more way to define a hash function is as the function that transforms a key into an address. The address generated by a hashing function is called the home address.



- Each letter of the word RANDY will be converted to its ASCII code. That just means we will turn each letter into a number. Then sum them all up, divide it with 100 and get the remainder.



KEY TERMS AND ISSUES in Hashing

Bucket –

- A bucket is an index position in a hash table that can store more than one record. Tables 11.1 and 11.2 show a bucket of size 1 and size 2, respectively.
- When the same index is mapped with two keys, both the records are stored in the same bucket. The assumption is that the buckets are equal in size.

Table 11.1 Table with bucket size 1

Index	Bucket of size 1
0	Alka
1	Bindu
2	
3	Deven
4	Ekta
5	
6	Govind
⋮	⋮
13	Monika
⋮	⋮
18	Sharmila
⋮	⋮
25	Zinat

Table 11.2(a) Table with bucket size 2

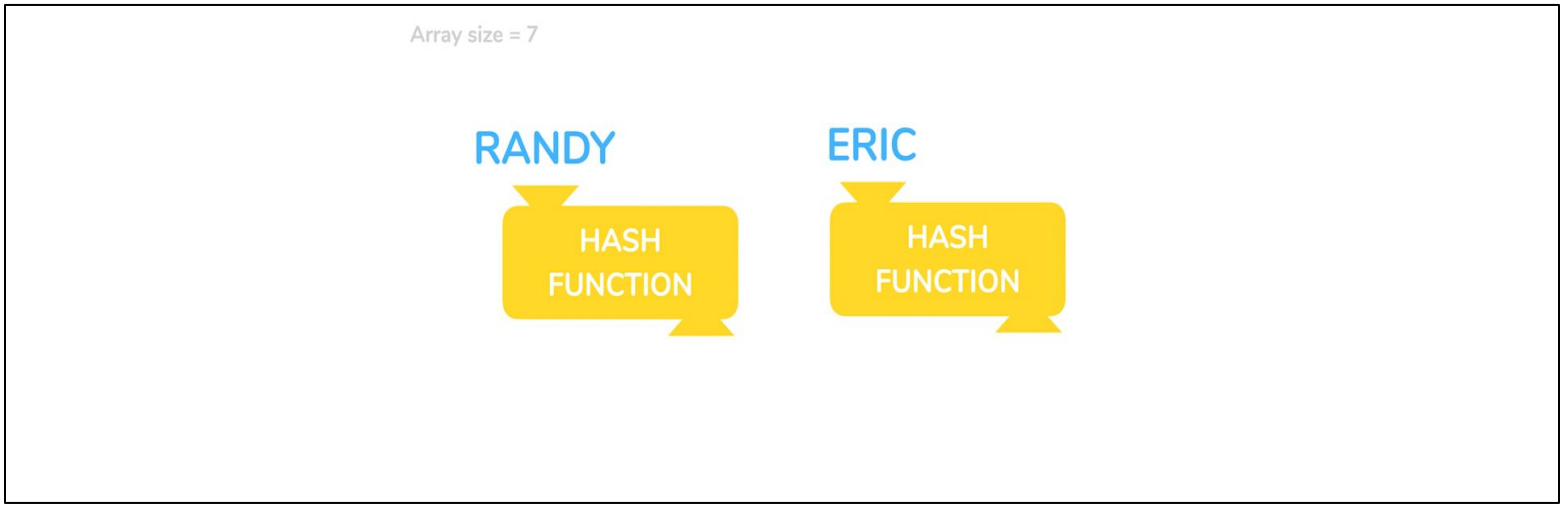
Index	Bucket of size 2	
0	Alka	Abhay
1	Bindu	Babali
2		
3	Deepa	Deven
4	Ekta	Esha
5		
6	Govind	Gopal
⋮	⋮	⋮
13	Monika	Meera
⋮	⋮	⋮
18	Sharmila	Sindhu
⋮	⋮	⋮
25	Zinat	Ziya

Table 11.2(b) Table with bucket size 3

Index	Bucket of size 3		
0	Alka	Abhay	Asmita
1	Bindu	Babali	Bhanu
2			
3	Deepa	Deven	Deepak
4	Ekta	Esha	Eshwar
5			
6	Govind	Gopal	Gautam
⋮	⋮	⋮	⋮
13	Monika	Meera	Manisha
⋮	⋮	⋮	⋮
18	Sharmila	Sindhu	Shilpi
⋮	⋮	⋮	⋮
25	Zinat	Ziya	Zeba

KEY TERMS AND ISSUES in Hashing

- **Probe** - Each action of address calculation and check for success is called as a probe.Probing in hash tables is a technique used to resolve **collisions** by finding alternative slots for inserting data.
- **Collision** - The result of two keys hashing into the same address is called collision.



KEY TERMS AND ISSUES in Hashing

- **Synonym** –Keys that hash to the same address are called synonyms.
- **Overflow** -The result of many keys hashing to a single address and lack of room in the bucket is known as an overflow.
- **Collision and overflow** are synonymous when the bucket is of size 1.
- **Open or external hashing** When we allow records to be stored in potentially unlimited space, it is called as open or external hashing.
- **Closed or internal hashing** When we use fixed space for storage eventually limiting the number of records to be stored, it is called as closed or internal hashing.

KEY TERMS AND ISSUES in Hashing

- **Hash function -** Hash function is an arithmetic function that transforms a key into an address which is used for storing and retrieving a record.
- **Perfect hash function** The hash function that transforms different keys into different addresses is called a perfect hash function. The worth of a hash function depends on how well it avoids collision.
- **Load density** - The maximum storage capacity, that is, the maximum number of records that can be accommodated, is called as loading density.
- **Full table -** A full table is one in which all locations are occupied. Owing to the characteristics of hash functions, there are always empty locations, rather a hash function should not allow the table to get filled in more than 75%.

KEY TERMS AND ISSUES in Hashing

- **Load factor -**Load factor is the number of records stored in a table divided by the maximum capacity of the table, expressed in terms of percentage.
- **Rehashing -** Rehashing is with respect to closed hashing. When we try to store the record with Key1 at the bucket position Hash(Key1) and find that it already holds a record, it is collision situation. To handle collision, we use a strategy to choose a sequence of alternative locations Hash1(Key1), Hash2(Key1), and so on within the bucket table so as to place the record with Key1. This is known as rehashing.

KEY TERMS AND ISSUES in Hashing

Issues in hashing –

In case of collision, there are two main issues to be considered:

1. We need a good hashing function that minimizes the number of collisions.

2. We want an efficient collision resolution strategy so as to store or locate synonyms.

HASH FUNCTIONS

- To store a record in a hash table, a hash function is applied to the key of the record being stored, returning an index within the range of the hash table.
- The record is stored at that index position, if it is empty.
- With direct addressing, a record with key K is stored in slot K.
- With hashing, this record is stored at the location Hash(K), where Hash(K) is the function.
- The hash function Hash(K) is used to compute the slot for the key K. Let us discuss some issues regarding the design of good hash functions and also study the schemes for their creation.

Good Hash Function

- Performance of hashing depends on how the hash function distributes the set of keys among the slots.
- A good hash function is one which satisfies an assumption that any given record is equally likely to hash into any of the slots, independent of whether any other record has been already hashed to it or not.
- This assumption is known as **simple uniform hashing**.
- If the probability that a key 'Key' occurs in our collection is P(Key), and for M slots in our hash table, a uniform hashing function, Hash(Key), should ensure that for $0 \leq Key \leq M - 1$, $S\ P(Key) = 1$, are all equiprobable with probability $1/M$.
- The hash function should ensure that they are hashed to different locations.

Features of a Good Hashing Function

1. Addresses generated from the key are uniformly and randomly distributed.

2. Small variations in the value of the key will cause large variations in the record addresses to distribute records (with similar keys) evenly.

3. The hashing function must minimize the occurrence of collision.

Methods of implementing hash functions

Division Method

- One simple choice for a hash function is to use the modulus division indicated as MOD (the operator % in C/C++).
- The function MOD returns the remainder when the first parameter is divided by the second parameter.
- The result is negative only if the first parameter is negative and the parameters must be integers.
- The function returns an integer. If any parameter is NULL, the result is NULL.

Hash[Key] = Key % M

Methods of implementing hash functions

Division Method

- Key is divided by some number M, and the remainder is used as the hash address.
- This function gives the bucket addresses in the range of 0 through (M - 1), so the hash table should at least be of size M.
- The choice of M is critical.
- Binary keys of length in powers of two are usually avoided.
- A good choice of M is that it should be a prime number greater than 20.

Methods of implementing hash functions - Multiplication Method

This method involves the following steps:

1. Choose a constant value A such that 0 < A < 1.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:

$$h[K] = floor\ [M\ [kA\ mod\ 1]]$$

Here,

M is the size of the hash table.

k is the key value.

A is a constant value.

Example:

k = 12345

A = 0.357840

M = 100

h[12345] = floor[100 [12345* 0.357840 mod 1]]

= floor[100 (4417.5348 mod 1)]

= floor[100 (0.5348)]

= floor[53.48]

= 53

Example –

k = 1234

A = 0.35784

M = 100

So,

h (1234) = floor [100(1234 x 0.35784 mod 1)]

= floor [100 (441.57456 mod 1)]

= floor [100 (0. 57456)]

= floor [57.456]

= 57

Methods of implementing hash functions

Extraction Method

- When a portion of the key is used for address calculation, the technique is called as the extraction method.
- In digit extraction, a few digits are selected, extracted from the key and are used as the address.

Key	Hashed address
345678	357
234137	243
952671	927

- If the portion of the key is carefully selected, it can be sufficient for hashing, provided the remaining portion distinguishes the keys in an insufficient way.

Methods of implementing hash functions

Mid-square Hashing

- Mid-square hashing suggests to take the square of the key and extract the middle digits of the squared key as the address.
- The difficulty is when the key is large.
- As the entire key participates in the address calculation, if the key is large, then it is very difficult to store its square as it should not exceed the storage limit.
- So mid-square is used when the key size is less than or equal to 4 digits.
- If the key is a string, it has to be preprocessed to produce a number.

Key	Square	Hashed address
2341	5480281	802
1671	2792241	922

Methods of implementing hash functions

Mid-square Hashing

- The difficulty of storing the squares of larger numbers can be overcome if we use fewer digits of the key (instead of the whole key) for squaring.
- If the key is large, we can select a portion of the key and square it.

Key	Square	Hashed address
234137	$234 \times 234 = 54756$	475
567187	$567 \times 567 = 321489$	148

Folding Technique

- In this technique, the key is subdivided into subparts that are combined or folded and then combined to form the address.
- For a key with digits, we can subdivide the digits into three parts, add them up, and use the result as an address.
- Here the size of the subparts of the key is the same as that of the address.

Methods of implementing hash functions - Folding Technique

There are two types of folding methods:

1.**Fold shift**—Key value is divided into several parts of the size of the address. Left, right, and middle parts are added.

2. **Fold boundary**—Key value is divided into parts of the size of the address. Left and right parts are folded on the fixed boundary between them and the center part.

For example, if the key is **987654321,** it is understood as

Left - 987 **Centre - 654** **Right - 321**

For fold shift, the sum is **987 + 654 + 321 = 1962**.

Now discard digit 1 and the address is **962.**

For fold boundary, **sum of the reverse** of the parts is **789 + 456 + 123 = 1368.**

Discard digit 1 and the address is **368.**

Methods of implementing hash functions

Rotation

- When the keys are serial, they vary only in the last digit and this leads to the creation of synonyms.
- Rotating the key would minimize this problem.
- This method is used along with other methods.
- Here, the key is rotated right by one digit and then folding technique is used to avoid synonyms.
- For example, let the key be 120605, when it is rotated we get 512060. Then the address is calculated using any other hash function.

Methods of implementing hash functions

Universal Hashing

- Sometimes wrong operations are performed deliberately, such as choosing N keys all of which hash to the same slot, yielding an average retrieval time of O(n).
- Any fixed hash function is helpless to this sort of worst-case behavior.
- The only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored.
- It is a family of hash functions that can be efficiently computed by using a randomly selected hash function from a set of hash functions.
- This approach is called universal hashing and yields good performance on the average, no matter what keys are chosen.

Methods of implementing hash functions

Universal Hashing

- The main idea behind universal hashing is to select the hash function at random at runtime from a carefully designed set of functions.
- Because of randomization, the algorithm can behave differently on each execution; even for the same input.
- This approach guarantees good average case performance, no matter what keys are provided as input.

Hashing Examples

1. Write the output table for these key and value pair to insert in an array using division method and multiplication method you need to store values (arranged in a key-value pair) inside a hash table with 30 cells. Take value of A = 0.456

Input - (3, 21) (1, 72) (40, 36) (5, 30) (11, 44) (15, 33) (18, 12) (16, 80) (38, 99)

Key-Value	Hash Value (Division Method)	Hash Value (Multiplication Method with A = 0.456)
(3,21)		
(1,72)		
(40,36)		
(5,30)		
(11,44)		
(15,33)		
(18,12)		
(16,80)		
(38,99)		

Key-Value	Hash Value (Division Method)	Hash Value (Multiplication Method with A = 0.456)
(3,21)	3	13
(1,72)	1	1
(40,36)	10	20
(5,30)	5	7
(11,44)	11	18
(15,33)	15	23
(18,12)	18	27
(16,80)	16	24
(38,99)	8	2

COLLISION RESOLUTION STRATEGIES

- No hash function is perfect.
- If **Hash(Key1) = Hash(Key2),** then Key1 and Key2 are synonyms and if bucket size is 1, we say that collision has occurred.
- As a consequence, we have to store the record Key2 at some other location.
- A search is made for a bucket in which a record is stored containing Key2, using one of the several collision resolution strategies.

COLLISION RESOLUTION STRATEGIES

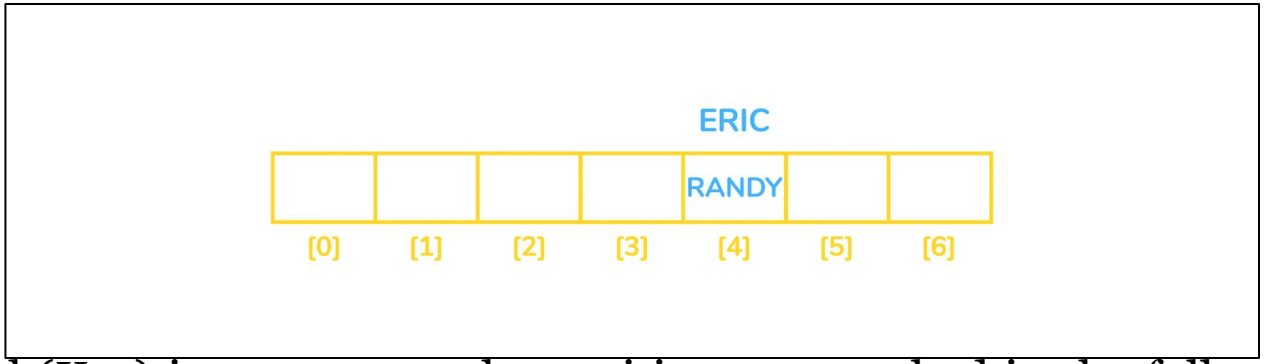
The collision resolution strategies are as follows:

1. **Open addressing**
    1. Linear probing
    2. Quadratic probing
    3. Double hashing
    4. Key offset
2. **Separate chaining (or linked list)**
3. **Bucket hashing (Closed Addressing) -** The most important factors to be taken care of to avoid collision are the table size and choice of the hash function.

COLLISION RESOLUTION STRATEGIES

Open Addressing

- In open addressing, when collision occurs, it is resolved by finding an available empty location other than the home address.



- If Hash(Key) is not empty, the positions are probed in the following sequence until an empty location is found.
- When we reach the end of table, the search is wrapped around to start and the search continues till the current collision location.

COLLISION RESOLUTION STRATEGIES

Open Addressing

N(Hash(Key) + C(1)), N(Hash(Key) + C(2)), …, N(Hash(Key) + C(i)), …

- Here **N** is the normalizing function
- **Hash(Key)** is the hashing function, and
- **C(i)** is the collision resolution (or probing) function with the i^{th} probe.
- The normalizing function is required when the resulting index is out of range.
- A commonly used normalization function is **MOD**.

COLLISION RESOLUTION STRATEGIES

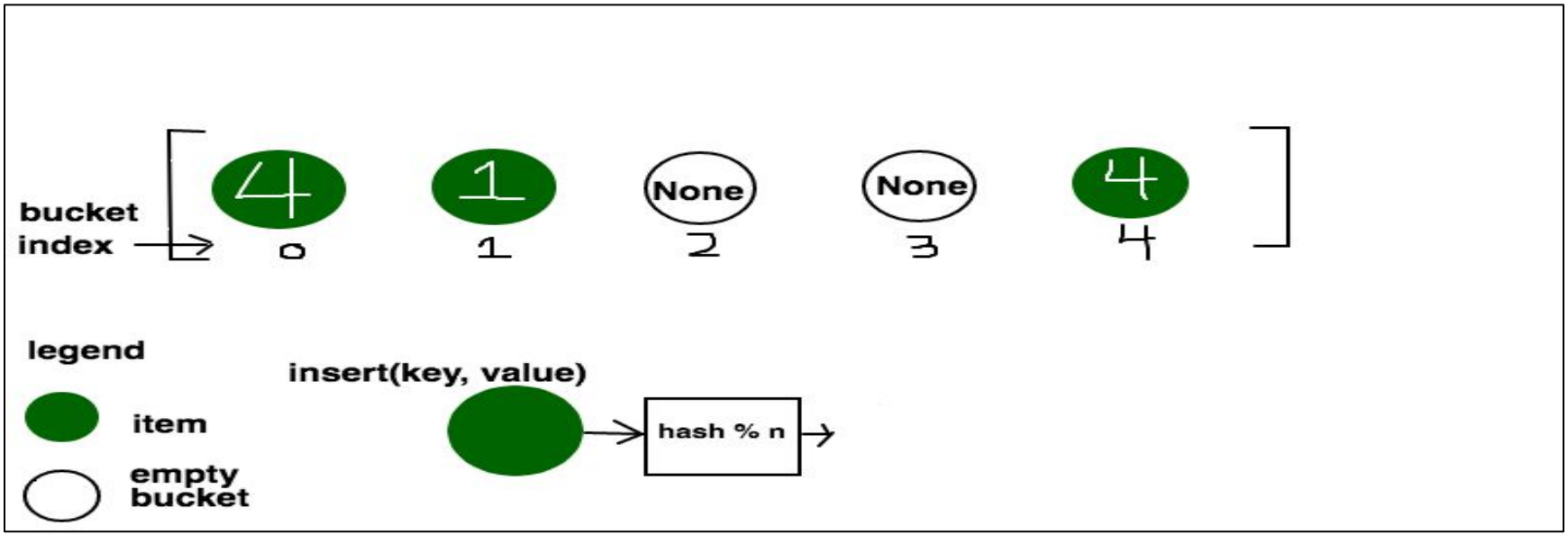
Open Addressing

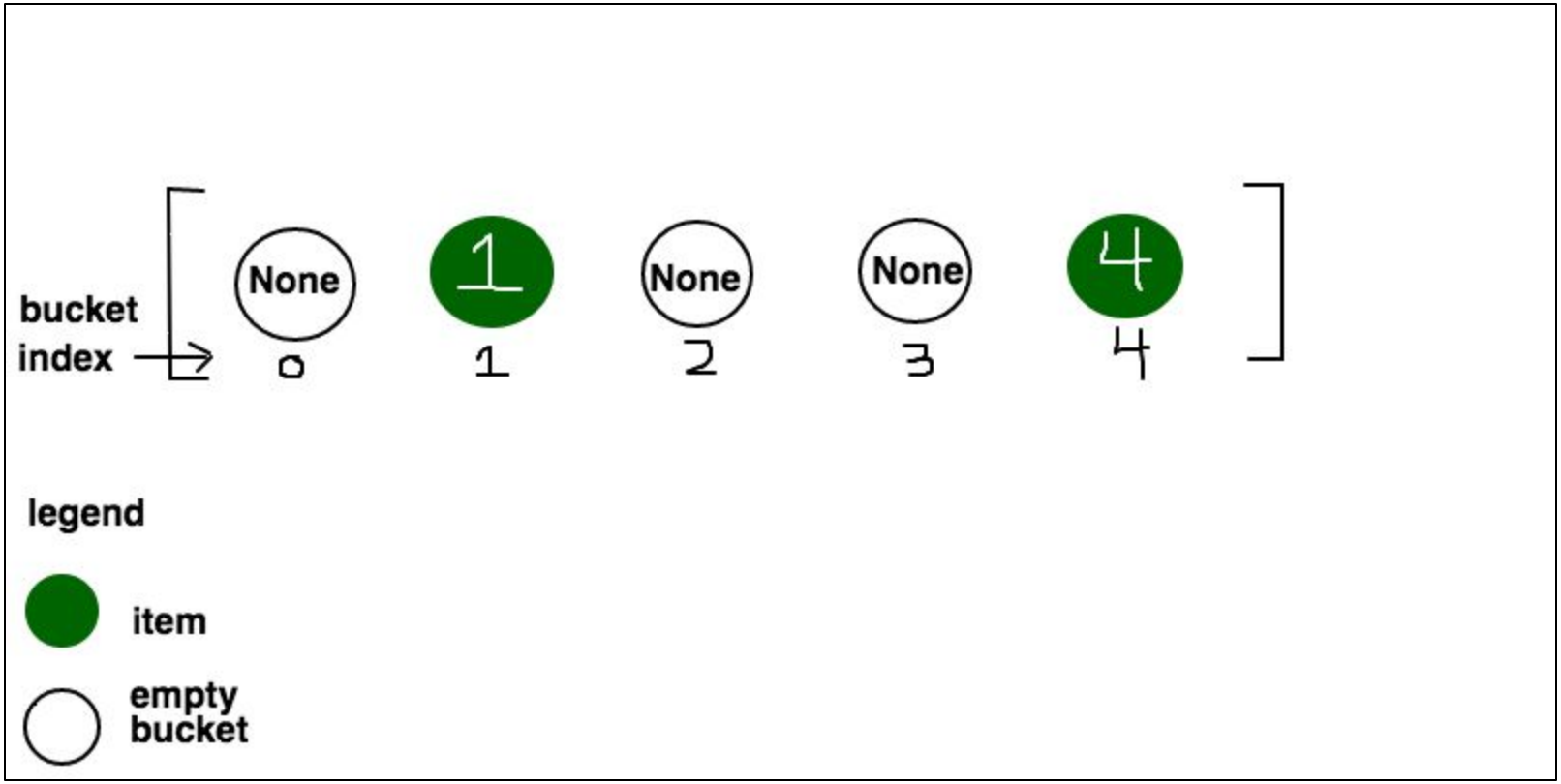
- In open addressing, when collision occurs, the table is searched for empty locations to store synonyms.
- Each table entry either contains a record or is empty.
- While searching for a record, we systematically examine table slots until the desired record is found or it is clear that the record is not in the table.
- While open addressing, to store the record, we successively examine, or probe, the hash table until we find an empty slot.
- Three techniques are commonly used to compute the probe sequences required for open addressing—**linear probing, quadratic probing, and rehashing**.

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

- A hash table in which a collision is resolved by placing the item in the next empty place following the occupied place is called linear probing. This strategy looks for the next free location until it is found.




bucket
index
None
1
None
None
4
0
1
2
3
4
legend
item
empty
bucket

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

- The function that we can use for probing linearly from the next location is as follows:

$$(\text{Hash}(x) + \text{C}(i)) \text{ MOD Max}$$

As $\text{C}(i) = i$ for linear probing in Eq. (11.1), the function becomes

$$(\text{Hash}(x) + i) \text{ MOD Max}$$

- Initially i = 1, if the location is not empty then it becomes 2, 3, 4, …, and so on till an empty location is found.
- We simply add one to the current address when collision occurs or till we find an empty location within the hash table limits.
- Here Max is the table size or the nearest prime number greater than the table size.
- The use of MOD wraps the linear probing to the table start, if it reaches the end.

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

Linear probing can be done using the following:

- **With replacement**—If the slot is already occupied by the key there are two possibilities, that is, either it is the home address (collision) or the location is occupied by some key. If the key's actual address is different, then the new key having the address at that slot is placed at that position and the key with the other address is placed in the next empty position.
- For example, in hash table of size 100, suppose Key1 = 127 is stored at address 25 and a new Key2 = 1325 is to be stored. Address for Key2 (1325 MOD 100) is 25. Now as the location 25 is occupied by Key1, the with replacement strategy places Key2 at location 25 and searches for an empty location for Key1 = 127.

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

2. Without replacement—When some data is to be stored in the hash table, if the slot is already occupied by the key, then another empty location is searched for a new record. There are two possibilities when the location is occupied—it is either its home address or not. In both the cases, the without replacement strategy searches for empty positions for the key that is to be stored .

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

Example

Store the following data into a hash table of size 10 and bucket size 1. Use linear probing for collision resolution. 12, 01, 04, 03, 07, 08, 10, 02, 05, 14 Assume buckets from 0 to 9 and bucket size = 1 using hashing function key % 10.

Solution - Let us use both techniques with and without replacement, as follows:

Linear probing with replacement - For linear probing with replacement, when collision occurs, if the location is occupied by a record whose home address is not that location, it is replaced and the current record is stored there.

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

Store the following data into a hash table of size 10 and bucket size 1. Use linear probing for collision resolution. 12, 01, 04, 03, 07, 08, 10, 02, 05, 14 Assume buckets from 0 to 9 and bucket size = 1 using hashing function key % 10.

Bucket	Initially empty	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	Insert 05	Insert 14
0								10	10	10	10
1			01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	12	12	12
3					03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	04
5									02	05	05
6										02	02
7						07	07	07	07	07	07
8							08	08	08	08	08
9											14

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

Linear probing without replacement - For linear probing without replacement when collision occurs, if the location is occupied, the next empty location is linearly probed for synonyms. Table 11.8 shows linear probing without replacement.

Bucket	Initially empty	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	Insert 05	Insert 14
0								10	10	10	10
1			01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	12	12	12
3					03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	04
5									02	02	02
6										05	05
7						07	07	07	07	07	07
8							08	08	08	08	08
9											14

COLLISION RESOLUTION STRATEGIES

Open Addressing – Linear Probing

```
def Hash(key,MAX):
    return key % MAX

def linear_p(index,MAX):
    i=1
    while(H[index]!=0):
        n_index=(index+i) % MAX
        if(H[n_index]==0):
            return n_index
        i=i+1
```

```
#main Program
MAX = 10
H =[0]*MAX
for i in range(0,MAX):
    key=int(input("Enter Key:"))
    index=Hash(key,MAX)
    if(H[index]==0):
        H[index]=key
    else:
        n_index=linear_p(index,MAX)
        H[n_index]=key
else:
    print("Overflow")
print(H)
```

COLLISION RESOLUTION STRATEGIES

Open Addressing – Quadratic Probing

- In quadratic probing, we add the offset as the square of the collision probe number. In quadratic probing, the empty location is searched by using the following formula:

$$(\text{Hash}(\text{Key}) + i^2) \text{ MOD Max where } i \text{ lies between 1 and } (\text{Max} - 1)/2$$

- Here if Max is a prime number of the form Gaussian prime(4 x integer + 3), quadratic probing covers all the buckets in the table.
- Quadratic probing works much better than linear probing, but to make full use of the hash table, there are constraints on the values of i and Max so that the address lies within the table boundaries.

Insert
18, 89, 21

0	
1	21
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert
58

	21
	58
	18
	89

For **58**:

- H = hash(58, 10) = 8
- Probe sequence:
  - i = 0, (8+0)% 10 = 8
  - i = 1, (8+1) % 10 = 9
  - i = 2, (8+4) % 10 = 2

Insert
68

	21
	58
	68
	18
	89

For **68**:

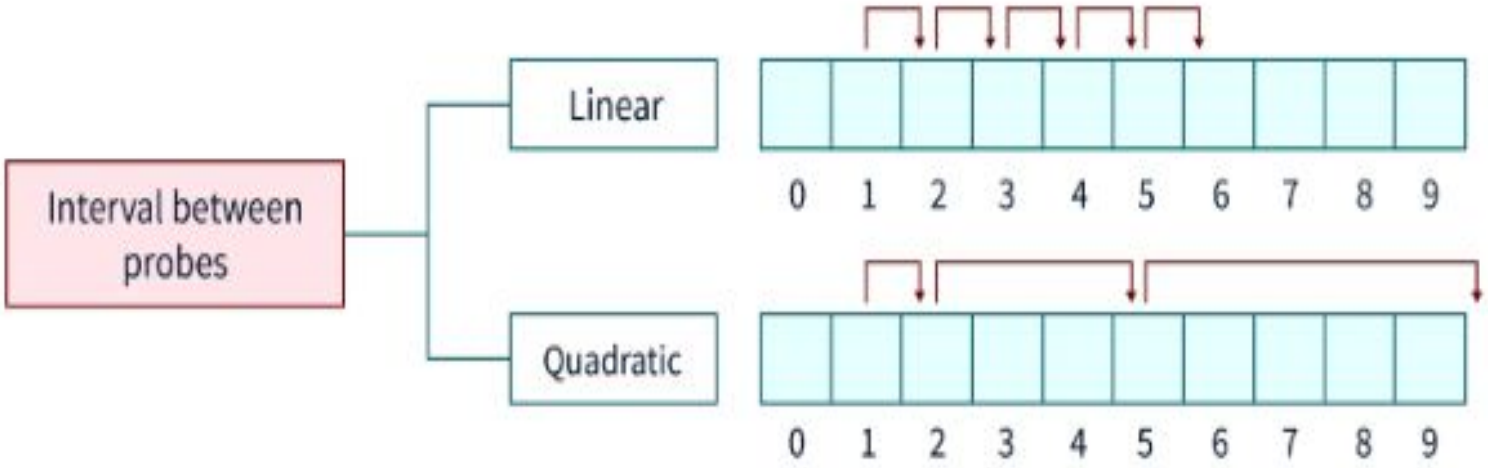
- H = hash(68, 10) = 8
- Probe sequence:
  - i = 0, (8+0)% 10 = 8
  - i = 1, (8+1) % 10 = 9
  - i = 2, (8+4) % 10 = 2
  - i = 3, (8+9) % 10 = 7

COLLISION RESOLUTION STRATEGIES

Open Addressing – Quadratic Probing

```
def Hash(key,MAX):

    return key % MAX

def quadratic_p(index, MAX):

    for i in range (MAX):

        #print("i=",i)

        n_index = (index + i ** 2) % MAX

        if H[n_index] == 0:

            print("index",n_index)

            return int(n_index)
```

```
# main Program
MAX = 10
H = [0] * MAX
print(H)
for i in range(0, MAX):
    key = int(input("Enter Key:"))
    index = Hash(key, MAX)
    if H[index] == 0:
        H[index] = key
    else:
# Change the probing function to quadratic_p
        n_index = quadratic_p(index, MAX)
        print(type(n_index))
        if H[n_index]==0:
            H[n_index] = key
  else:
    print("Overflow")
print(H)
```


Interval between probes
Linear
Quadratic
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

COLLISION RESOLUTION STRATEGIES

EXAMPLE

Suppose Max = 8 and keys A, B, C, D have hash values Hash(A) = 3, Hash(B) = 0, Hash(C) = 4, and Hash(D) = 3. Use linear probing for collision resolution.

Solution - Linear probing is the simplest strategy where Hash(Key) = Hash((Key + i) MOD Max). Suppose we wish to insert D and find that bucket 3 has been filled already, then we would try buckets 4, 5, 6, 7, 0, 1, and 2 in sequence. We find bucket 5 empty and we store D.

0	*B*
1	
2	
3	*A*
4	*C*
5	*D*
6	
7	

EXAMPLE

Consider the keys 22, 17, 32, 16, 5, and 24. Let Max = 7. Let us use quadratic probing to handle synonyms.

Solution - Let the hash functions be (Key MOD Max); for quadratic probing (Hash(Key) ± i ²) MOD Max. After storing 22, 17, 32, 5, and 7, the table looks as shown in the left column of Table

Index	Key		Index	Key
0			0	24
1	22		1	22
2	16	insert 24 →	2	16
3	17		3	17
4	32		4	32
5	5		5	5
6			6	

- We can see that while inserting 24, the address we get is Hash(24) = 24 MOD 7 = 3
- It is also noted that the location 3 is already occupied.
- We may now go for the quadratic function as [Hash(24)+(1)² MOD 7] = (24 MOD 7) + 1 MOD 7 = (3 + 1) MOD 7 = 4 which is not occupied.
- Hence,

  Hash(24) + (2)² MOD 7 = (3 + 4) MOD 7 = 0

  which is empty, so store 24 there.

COLLISION RESOLUTION STRATEGIES

Open Addressing – Double Hashing

- **Double hashing** uses two hash functions, one for accessing the home address of a Key and the other for resolving the conflict.
- The sequence for probing is generated as follows:

(Hash1(Key), (Hash1(Key) + i X Hash2(Key)), ...

i = 1, 2, 3, 4, … and the resultant address is modulo Max.

COLLISION RESOLUTION STRATEGIES

Open Addressing – Double Hashing

EXAMPLE - Let the hash function be Key % 10, Max = 10, and the keys be 12, 01, 18, 56, 79, 49. Perform double hashing.

Solution - Table demonstrates all insertions and collision handling using double hashing.

	Initially empty	Insert 12	Insert 01	Insert 18	Insert 56	Insert 79	Insert 49
0							
1			01	01	01	01	01
2		12	12	12	12	12	12
3							
4							
5							49
6					56	56	56
7							
8				18	18	18	18
9						79	79

- While inserting 49, the hashed location 9 is found occupied by key 79, so let us use ***Hash2(key) = R - (Key MOD R),*** where R is a small prime number, even smaller than the table size.
- **Let us use R = 7.**
- To insert 49, using **Hash1(Key) = 49 % 10**, we get 9 which is already occupied, so we use Hash2 as follows: **Hash2(49) = 7 - (49 % 7) =7 - 0 = 7**
- Hence by double hashing, **Hash(49) = [Hash1(49) + Hash2(49)] % 10 = (9 + 7) % 10 = 6** and location 6 is not empty, so let us recompute again.
- **Hash(49) = [Hash1(49) + 2 X Hash2(49)] % 10 = 9 + 2 X 7 = 23 % 10 = 3** and is empty, so store key 49 there.

EXAMPLE - Given the input {4371, 1323, 6173, 4199, 4344, 9699, 1889} and hash function as Key % 10, show the results for the following:
1. Open addressing using linear probing
2. Open addressing using quadratic probing
3. Open addressing using double hashing h2(x) = 7 - (x MOD 7)

Solution - The results are as follows:
1**. Open addressing using linear probing** These keys are inserted using linear probing as shown in Table

	Initially empty	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							9699	9699
1		4371	4371	4371	4371	4371	4371	4371
2								1889
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5						4344	4344	4344
6								
7								
8								
9					4199	4199	4199	4199

2. **Open addressing using quadratic probing -** Let us insert these keys using quadratic probing now as shown in Table

	Initially empty	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							9699	9699
1		4371	4371	4371	4371	4371	4371	4371
2								
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5						4344	4344	4344
6								
7								
8								1889
9					4199	4199	4199	4199

3. Open addressing using double hash function - Table shows the status of the hash table after inserting each key using open addressing using double hashing

	Initially empty	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0								1889
1		4371	4371	4371	4371	4371	4371	4371
2							9699	9699
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5								
6								
7						4344	4344	4344
8								
9					4199	4199	4199	4199

COLLISION RESOLUTION STRATEGIES

REHASHING

- If the table gets full, insertion using closed hashing with quadratic probing might fail or it might take too much time.
- The solution for this problem is to build another table that is about twice as big and scan down the entire original hash table, compute the new hash value for each record, and insert them in a new table.
- For example, if initially, the table is of size 7 and the hash function is key % 7 then, this would be as shown in Table.

	Insert 7, 15, 13, 74, 73
0	7
1	15
2	
3	73
4	74
5	
6	13

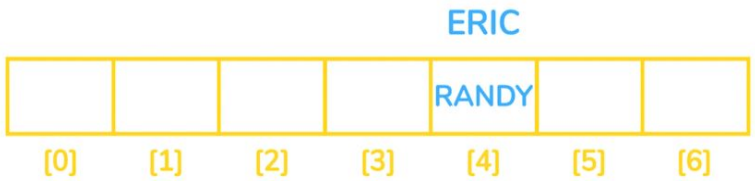
- As the table is more than 70% full, a new table is created. The size of the new table is 17, that is next prime of double of 7 that is 14.

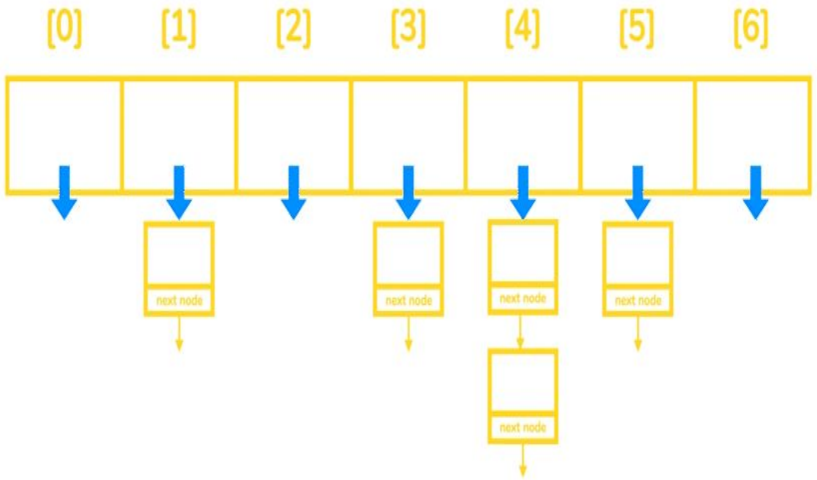
0	
1	
2	
3	
4	
5	73
6	74
7	7
8	
9	
10	
11	
12	
13	13
14	
15	15
16	

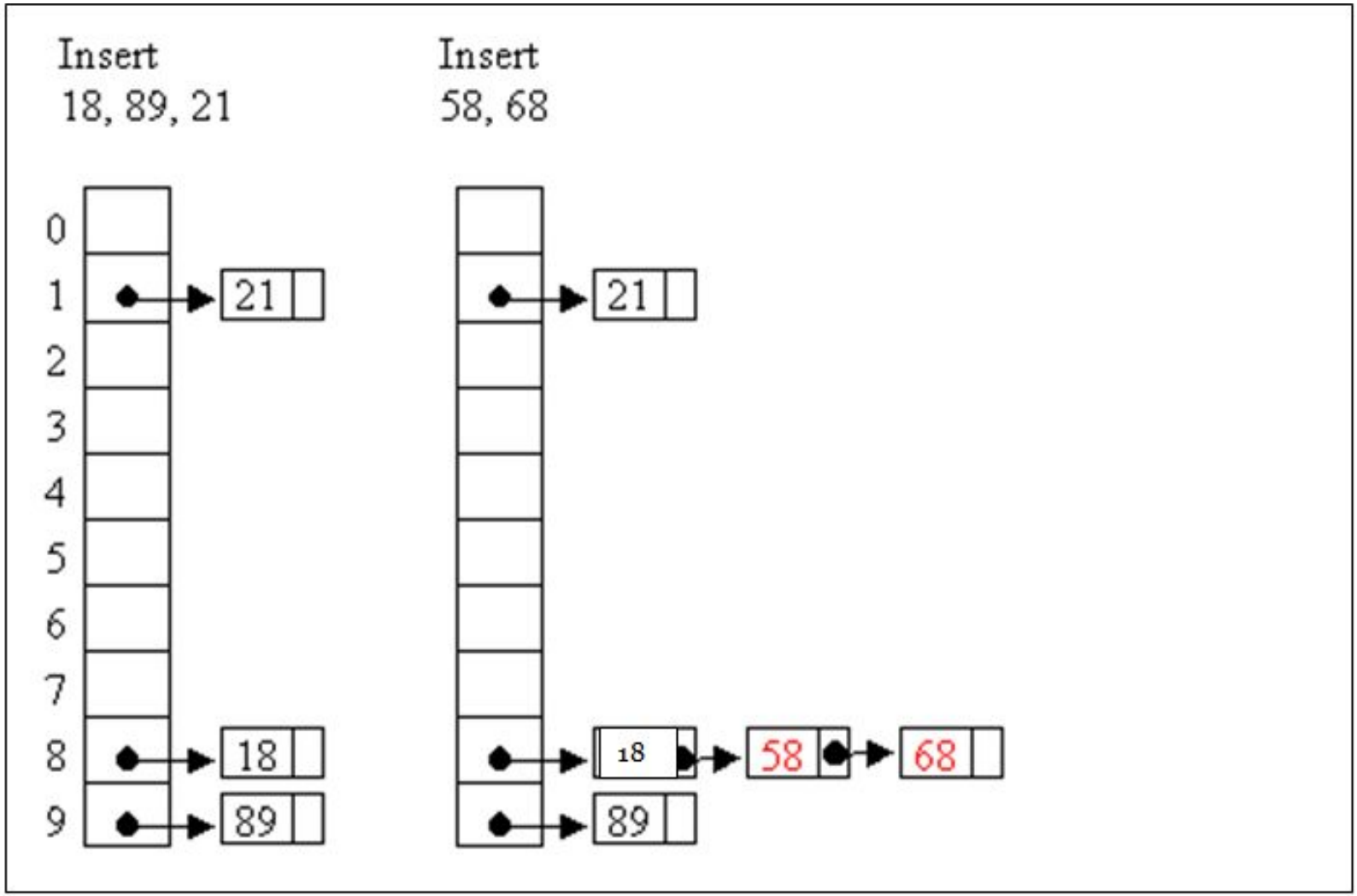
COLLISION RESOLUTION STRATEGIES

CHAINING

- The technique used to handle synonyms is chaining; it chains together all the records that hash to the same address.
- Instead of relocating synonyms, a linked list of synonyms is created whose head is the home address of synonyms.
- However, we need to handle pointers to form a chain of synonyms. The extra memory is needed for storing pointers.




[0]
[1]
[2]
[3]
[4]
[5]
[6]
next node
next node
next node
next node
next node


Insert
18, 89, 21
Insert
58, 68
0
1
2
3
4
5
6
7
8
9
21
18
89
21
18
58
68
89

In Fig., a hash table with Max = 10, both keys 322 and 262 probe to address 2. A chain, a linked list, stores all items at a particular home address

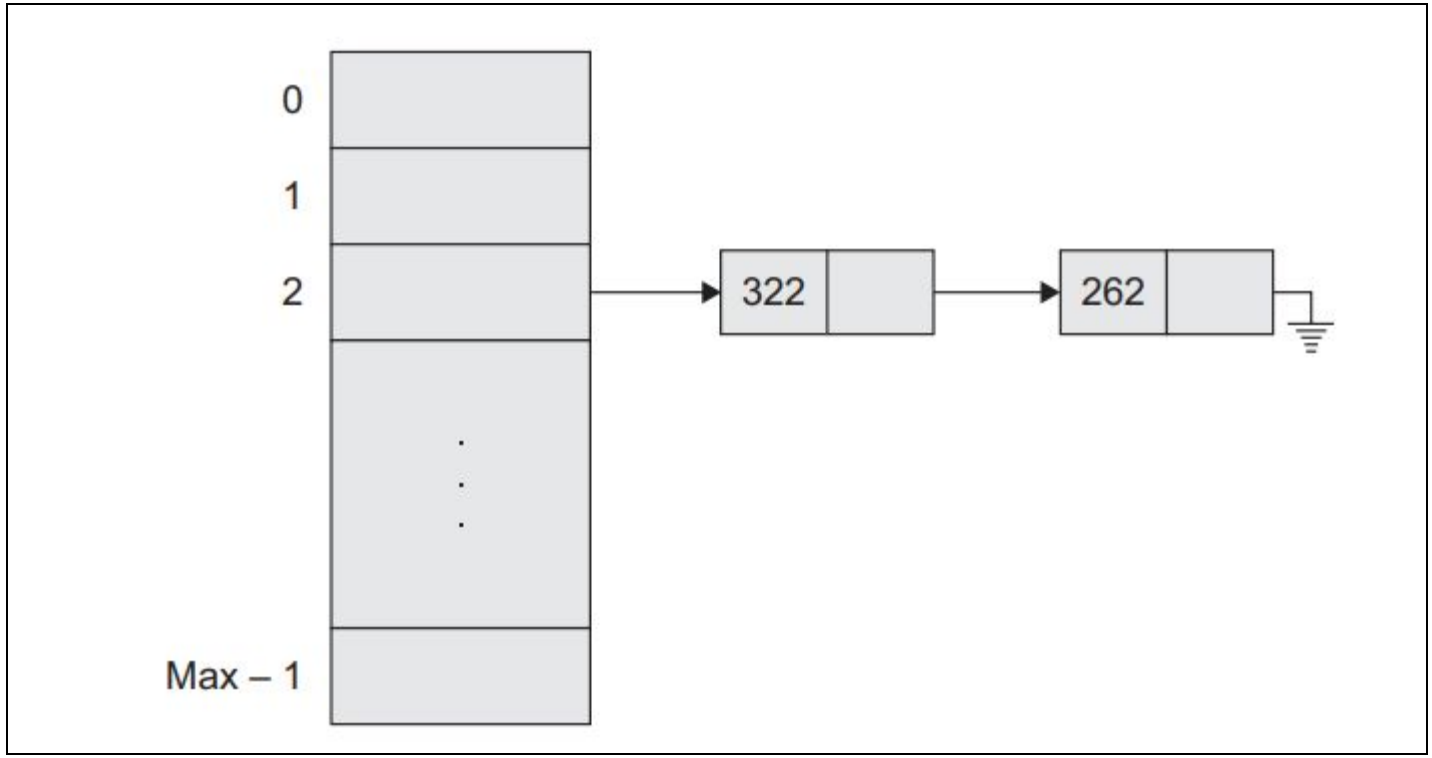


Table 11.16 Comparison of chaining and rehashing

Chaining	Rehashing
Unlimited number of synonyms can be handled.	A limited but good number of synonyms are taken care of.
Additional cost to be paid is an overhead of multiple linked lists.	The table size is doubled but no additional fields of links are to be maintained.
Sequential search through the chain takes more time.	Searching is faster when compared to chaining.

```
MAX = 10  # Define the maximum size of the hash table

class Node:
    def __init__(self, key):
        self.key = key
        self.next = None

hashtable = [None] * MAX #Initialize the hash table with empty lists

def init():
    """Initializes all slots in the hash table to None."""
    for i in range(MAX):
        hashtable[i] = None

def hash(key):
    """Hash function to get the position based on the key."""
    return key % MAX
```

```
def insert(k):
    """Inserts a key into the hash table using separate chaining."""
    pos = hash(k)
    new_node = Node(k)
    if hashtable[pos] is None:
        hashtable[pos] = new_node
    else:
        temp = hashtable[pos]
        while temp.next is not None:
            temp = temp.next
        temp.next = new_node
```

```
def display():
    """Displays the contents of the hash table."""
    for i in range(MAX):
        temp = hashtable[i]
        while temp is not None:
            print(temp.key, end="\t")
            temp = temp.next
        print()
```

```
def search(x):
 """Searches for a key in the hash
table."""
    pos = hash(x)
    temp = hashtable[pos]
    while temp is not None and temp.key !=
x:
        temp = temp.next
    if temp is None:
        print("Not Found")
    else:
        print("Key Found")
```

```
# Example usage
init()
insert(12)
insert(23)
insert(34)
insert(15)#Collision at index 5

display()
# Output:
# 12  15
# 23
# 34

search(23)  # Output: Key Found
search(45)  # Output: Not Found
```

HASH TABLE OVERFLOW

- Even if a hashing algorithm (function) is very good, it is likely that collisions will occur.
- The identifiers that have hashed into the same bucket, as discussed earlier, are called synonyms.
- An overflow is said to occur when a new identifier is mapped or hashed into a full bucket.
- When the bucket size is one ,a collision and an overflow occur simultaneously.
- There are a number of techniques for handling overflow of records

HASH TABLE OVERFLOW

Open Addressing for Overflow Handling

- In open addressing, we assume that the hash table is an array. When a new identifier is hashed into a full bucket, we need to find another bucket for this identifier. The simplest solution is to find the closest unfilled bucket through linear probing or linear open addressing.

HASH TABLE OVERFLOW

Open Addressing for Overflow Handling

When linear open addressing is used to handle overflows, a hash table search for an identifier I proceeds as follows:

1. Compute Hash(I)

2. Examine identifiers position

Table[Hash(I)], Table[Hash(I) + 1], …, Table[Hash[I] + i], in order until:

(a) *If Table[Hash(I) + j] = I then In this case I is found.*

(b) *If Table[Hash(I) + j] is NULL, then I is not in the table.*

(c) *If we return to the start position Hash(I), then the table is full and I is not in the table*.

HASH TABLE OVERFLOW

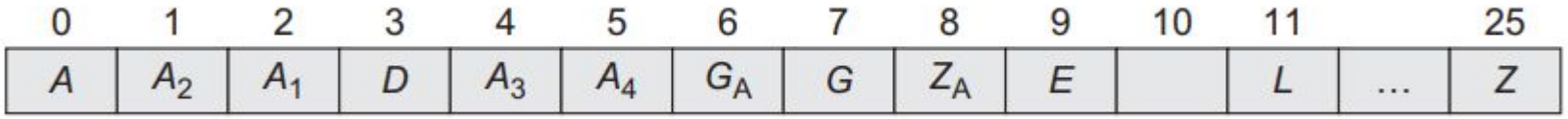
Open Addressing for Overflow Handling

- One of the problems with linear open addressing is that it tends to create clusters of identifiers.
- Moreover, these clusters tend to merge as more identifiers are entered, leading to big clusters.
- An alternative method to retard the growth of clusters is to use a series of hash functions h1, h2, …, hm.
- Buckets hi(x), $1 \leq i \leq m$ are examined in that order.

HASH TABLE OVERFLOW

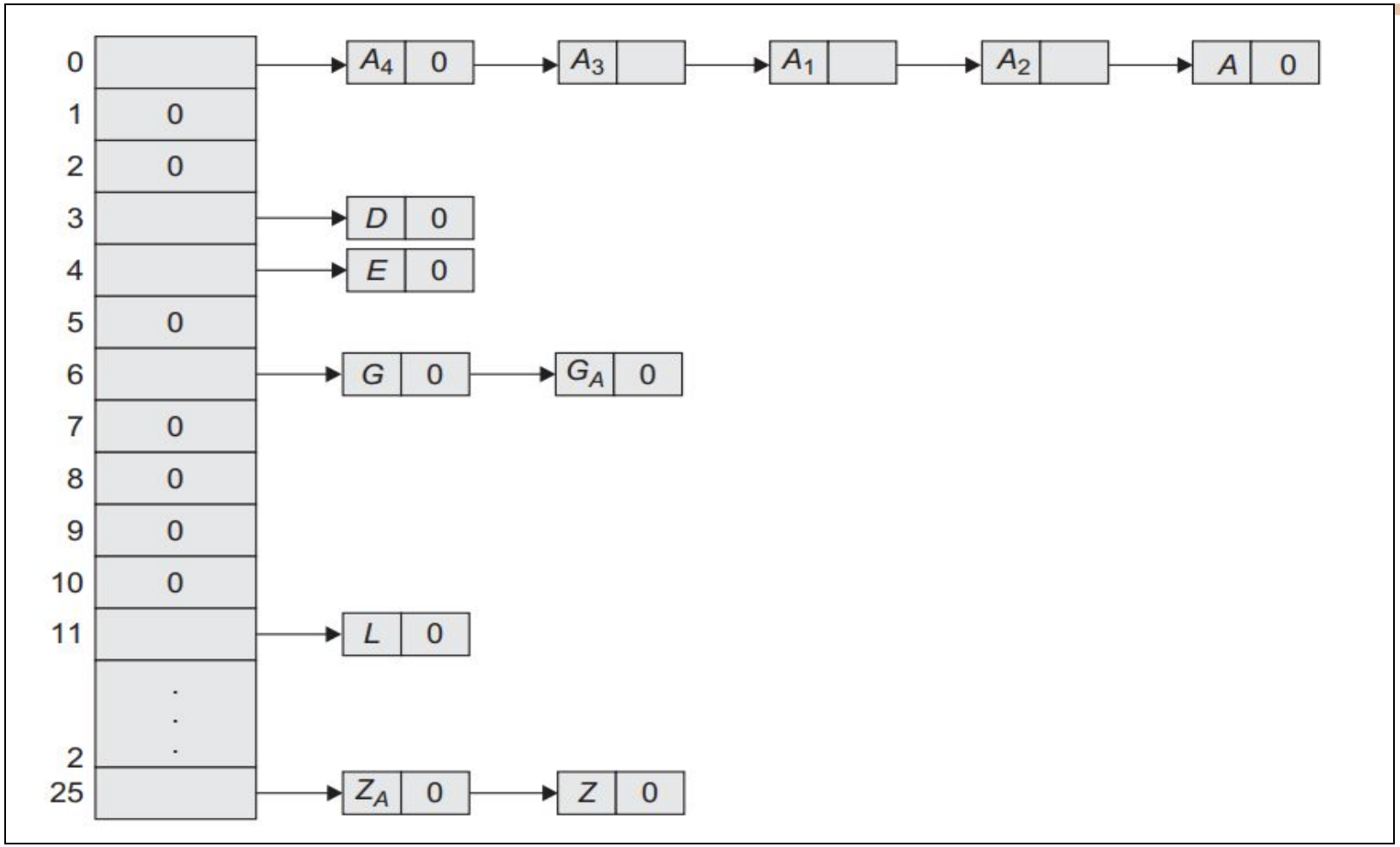
Overflow Handling by Chaining

- Consider the following hash table shown in Fig.



0	1	2	3	4	5	6	7	8	9	10	11		25
A	A_2	A_1	D	A_3	A_4	G_A	G	Z_A	E		L	...	Z

- In the above hash table of 25 buckets, one slot per bucket, searching for the identifier Z_A involves comparisons with the buckets Table[0] to Table[7
- Many of the comparisons can be saved if we maintain lists of identifiers, one list per bucket, each list containing all the synonyms for that bucket.
- If this is done, a search involves computing the hash address Hash(I) and examining only those identifiers in the list for Hash(I).
- Since the sizes of these lists are not known in advance, the best way to maintain them is as linked chains.

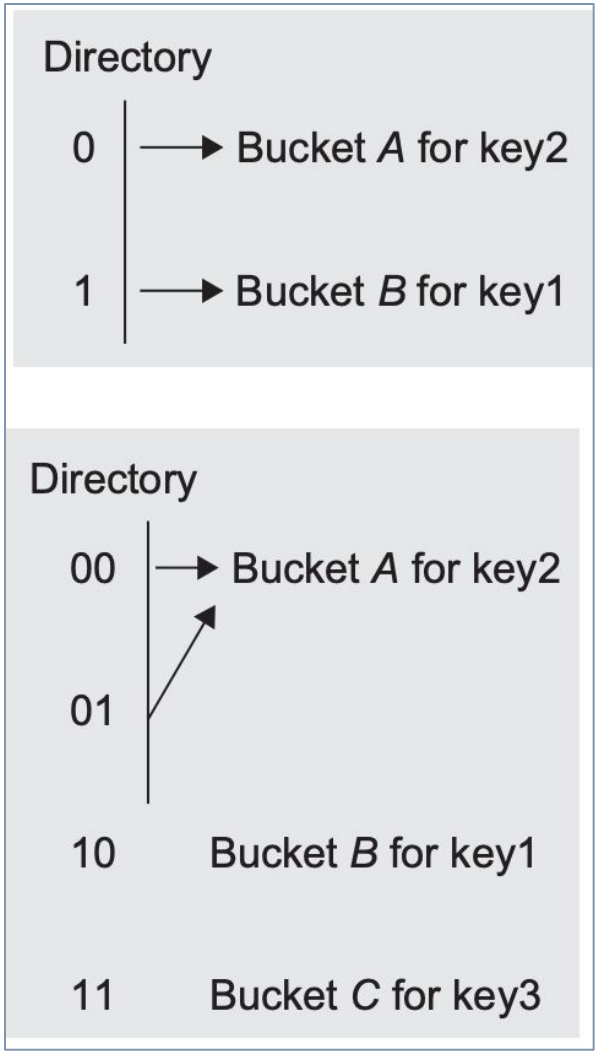


- To insert a new identifier, I, into a chain, we must first verify that it is not currently in chain. Then, if not present, I is inserted at any position in the chain.

EXTENDIBLE HASHING

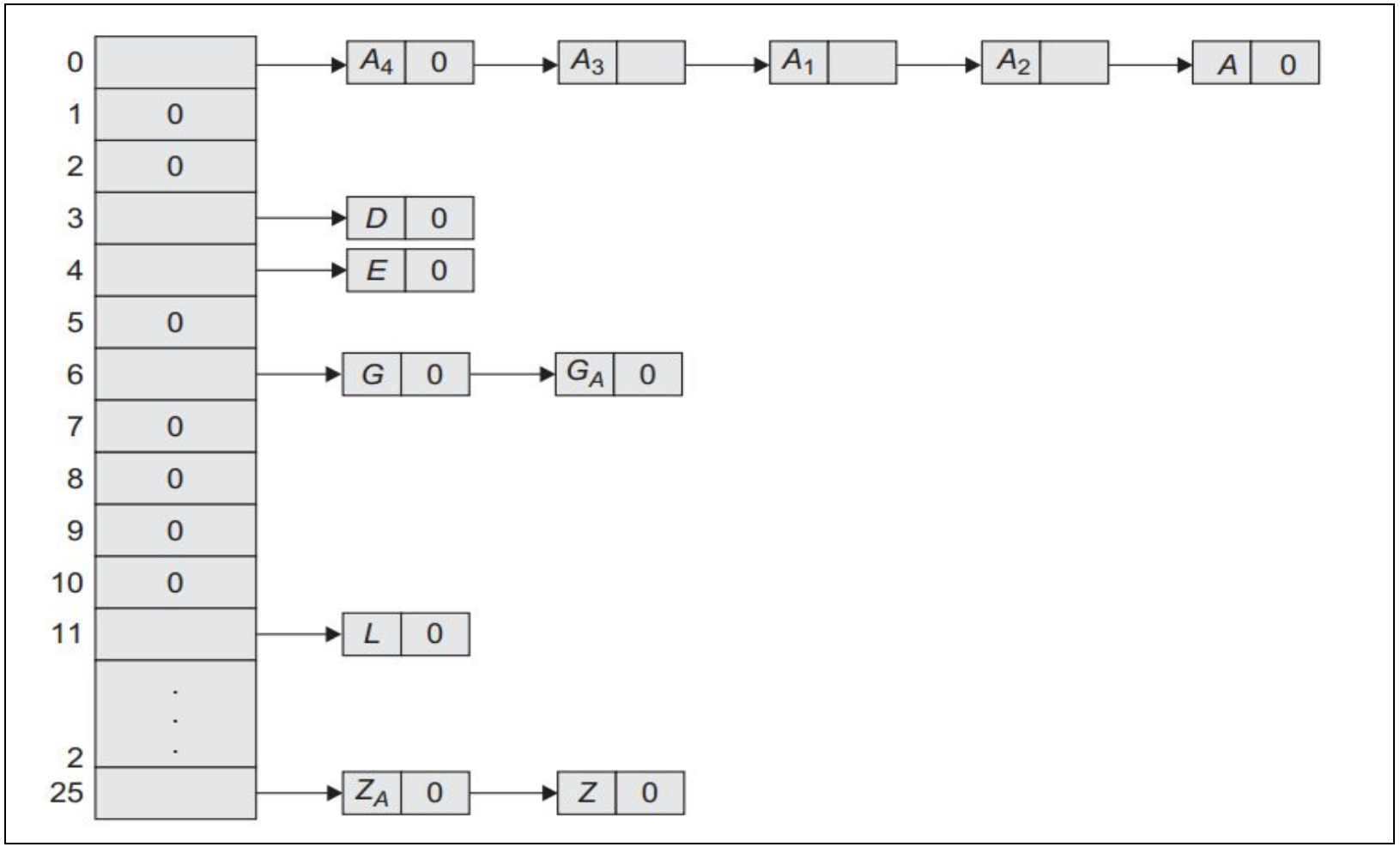
- For fast searching and less disk access, extendible hashing is used.
- Extendible hashing is a dynamic approach to managing data.
- This method is flexibile so that even the hashing function dynamically changes according to the situation and data type.
- It is a type of hash system, which treats a hash as a bit string, and uses a trie for bucket lookup.
- The first i bits of each string will be used as indices to figure out where they will go in the hash table. Additionally, i is the smallest number such that the first i bits of all keys are different.

- For example, assume that the hash function Hash(Key) returns a binary number.
- The first i bits of each string will be used as indices to figure out where they will go in the hash table. Additionally, i is the smallest number such that the first i bits of all keys are different.
- The keys to be used are as follows:
- 1. h(key1) = 100101
- 2. h(key2) = 011110
- 3. h(key3) = 110110



Closed Addressing

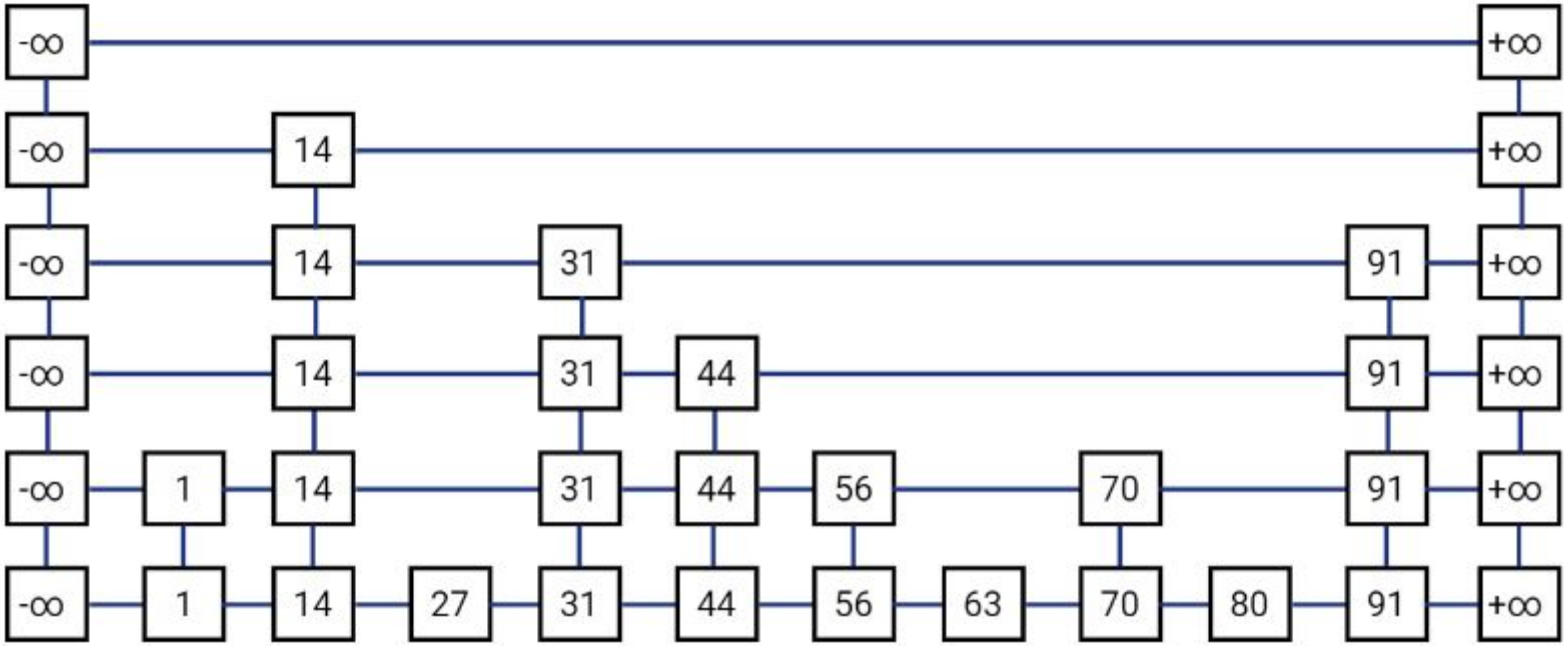
- **Closed-Addressing(Open Hashing) : *Closed-Address Hashing, also known as Open Hashing or Separate Chaining,*** is a hashing technique where each slot (bucket) in the hash table stores a linked list of elements that have the same hash value.
- When a collision occurs, meaning multiple elements are mapped to the same hash value, they are stored in the same slot using separate chaining.
- The advantage of Closed-Address Hashing is that it can handle a large number of elements in each slot, as the linked list can grow dynamically.
- It also ensures constant-time complexity for insertions and lookups in the average case.
- However, it may suffer from performance degradation when the linked lists become long, resulting in increased search time.


0
1
2
3
4
5
6
7
8
9
10
11
2
25
0
0
0
0
0
0
0
0
0
A_4 0
A_3
A_1
A_2
A 0
D 0
E 0
G 0
G_A 0
L 0
Z_A 0
Z 0

Skip List - Definition and Structure

- A skip list is a probabilistic data structure that contains a set of sorted linked lists.
- It is considered to be a good alternative to balanced search trees.
- A skip list is a multi-layer linked list.
- The bottom-most layer is a plain linked list that has elements in sorted order.
- Then there are several layers or levels above that, as additional linked lists, however, these additional layers have only a few important nodes which are chosen from the linked list of the last layer, in the same sorted order just like the bottom-most linked list.
- These additional layers form sort of an "express line" with the elements of the main list.
- The bottom-most linked list is called as "normal line".

- The problem of inefficient searching on a sorted linked list is solved by skip lists, which can skip over nodes while traversal.
- The core concept of a skip list is, here our nodes not only contain a single forward pointer as a traditional linked list but contain several pointers that point to different nodes(the only one which is in a forward direction from the current node) of the skip list.
- In addition, we use several methods to determine which forward pointer to choose while proceeding in the linked list.


-∞ +∞
-∞ 14 +∞
-∞ 14 31 91 +∞
-∞ 14 31 44 91 +∞
-∞ 1 14 31 44 56 70 91 +∞
-∞ 1 14 27 31 44 56 63 70 80 91 +∞

Types of Operations in the skip list

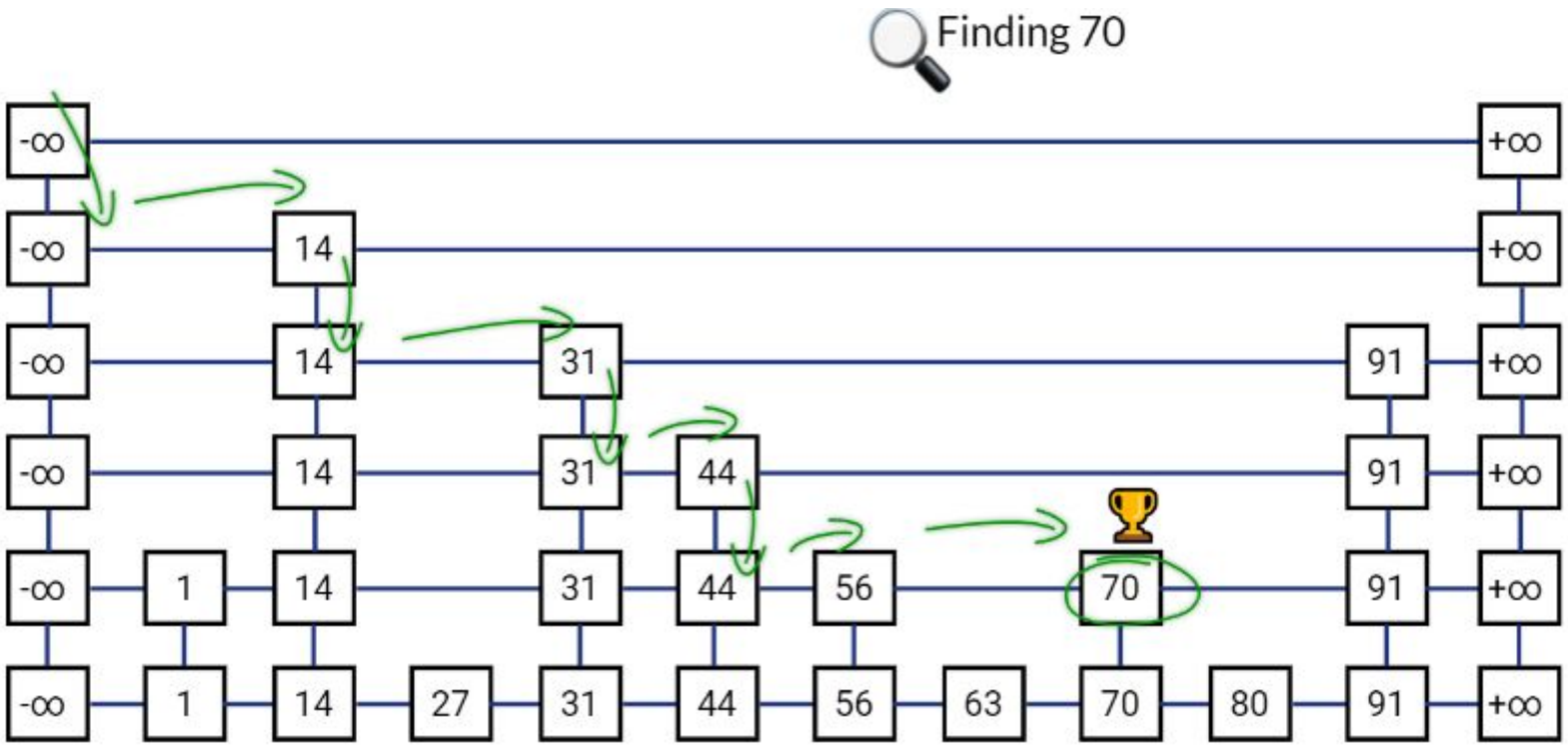
- A skip list, like a standard linked list, can perform the following operations, but skip lists allow faster insertion, deletion, and searching of elements than a linked list.
    - **Insertion Operation:** To add a new node to the skip list.
    - **Deletion Operation:** To delete a node from the skip list.
    - **Search Operation:** To search for a node in the skip list.

Search Operation:

- The process of searching from the top layer of our skip list. Let us break down the insertion algorithm in steps:
- We will iterate through the layer till we find an element that is just greater than the new node we are trying to insert.
- Once we find such a node we will use its pointer to go down to the lower layer.
- We will repeat steps 1 and 2 till we reach the bottom-most layer, once we reached the required node while performing step 1, do the insertion of the new node.
- A random level will be chosen for the inserted node and finally we have to rearrange a few pointers to reflect the insertion in the skip list.

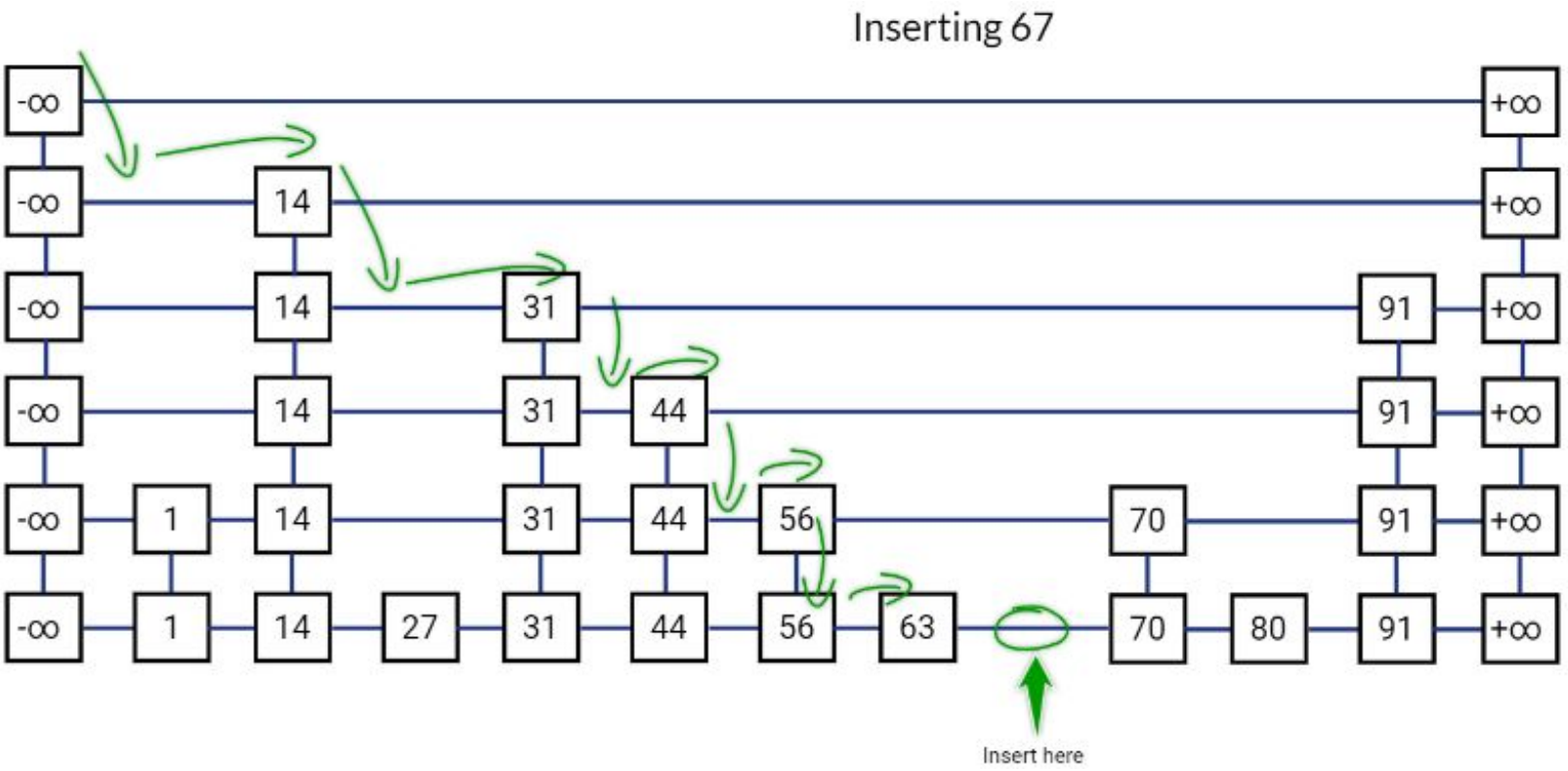
Search Operation:

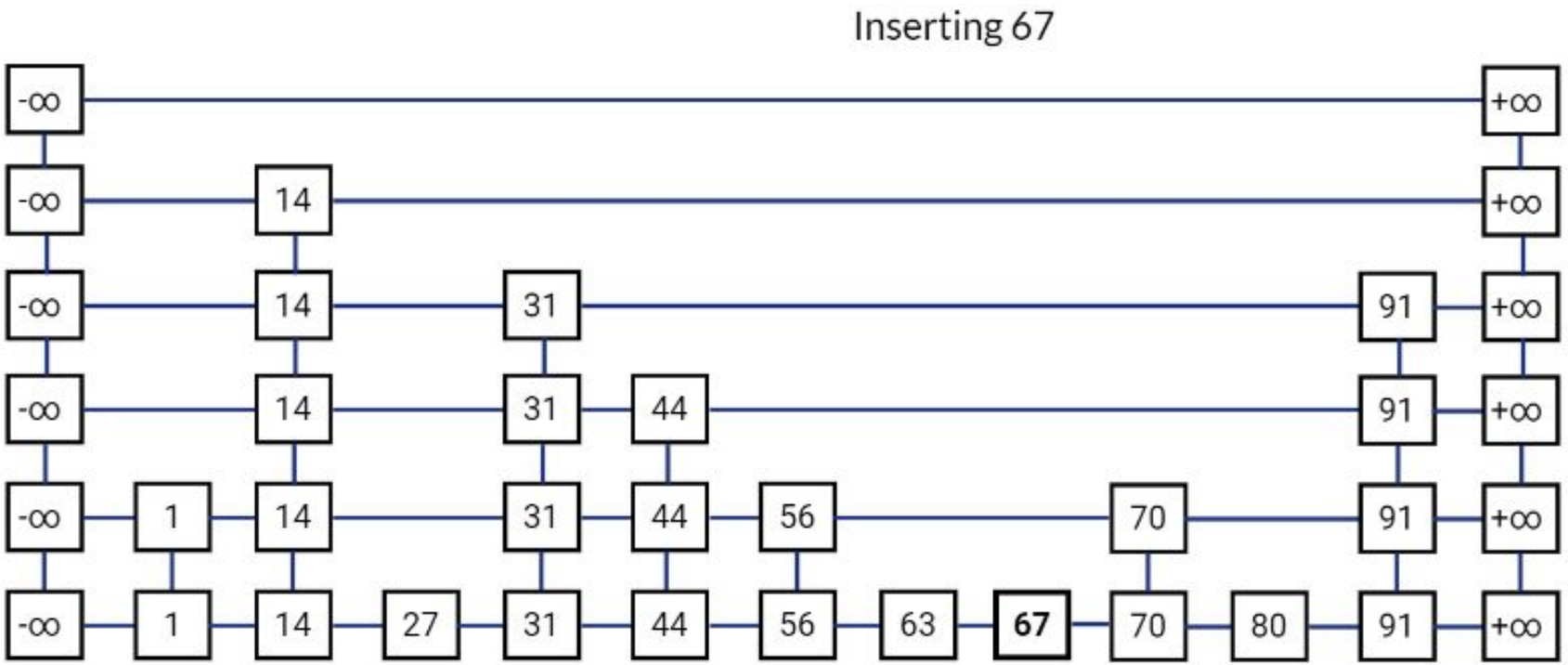
1. We will iterate through the layers, till we find an element that is just greater or equal to the node we are trying to find.
2. If any node is found that is equal to the given searched value, it would end the search. This is because the searched node has been found in one of the express lines itself.
3. Otherwise, we will use the current node's pointer to go down to the next layer.
4. We will repeat steps 1-3 till we reach the bottom-most layer in the worst case, print the node once encountered during the iteration on the bottom-most layer.
5. If the element doesn't exist in the bottom-most layer also then it implies that the element doesn't exist in the skip list.

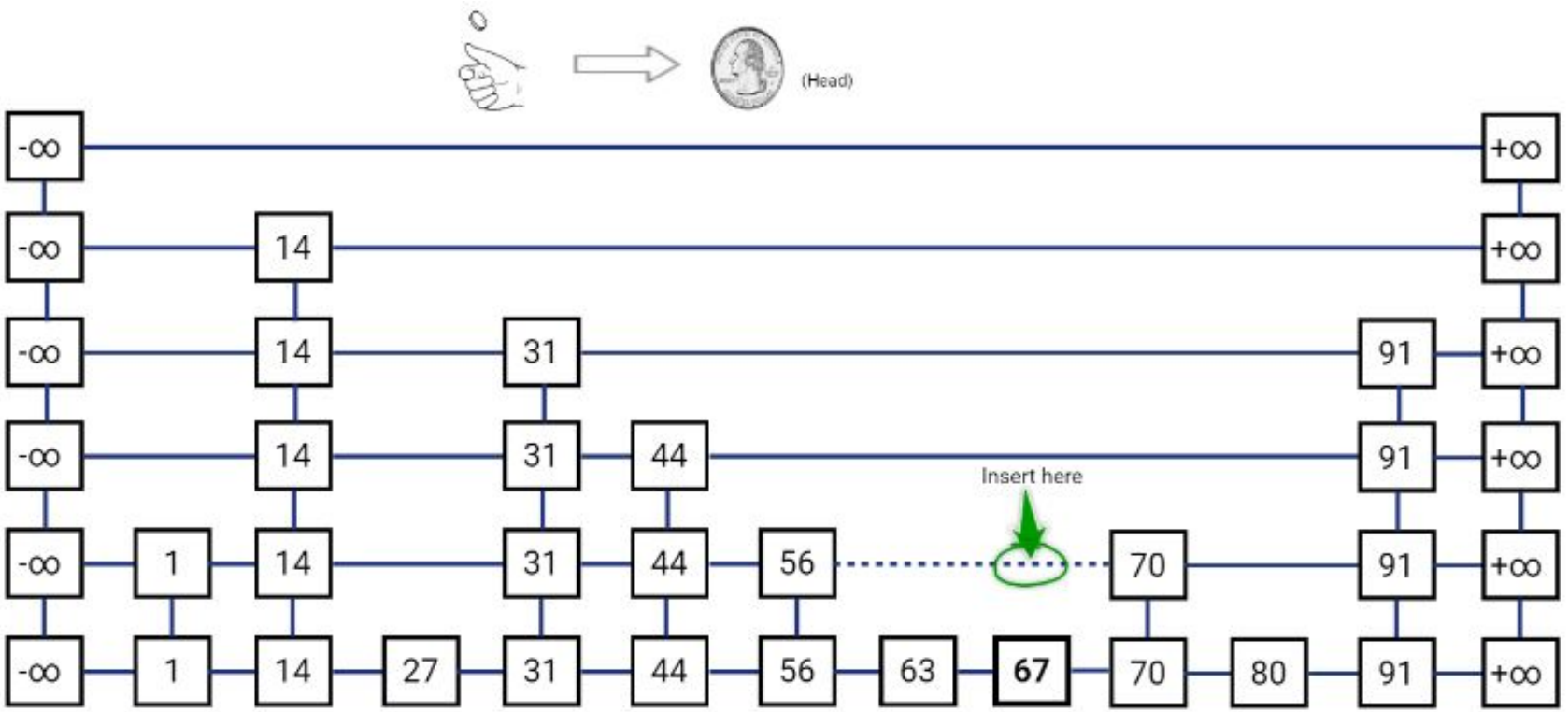

Finding 70
-∞
+∞
14
31
91
44
1
56
70
27
63
80

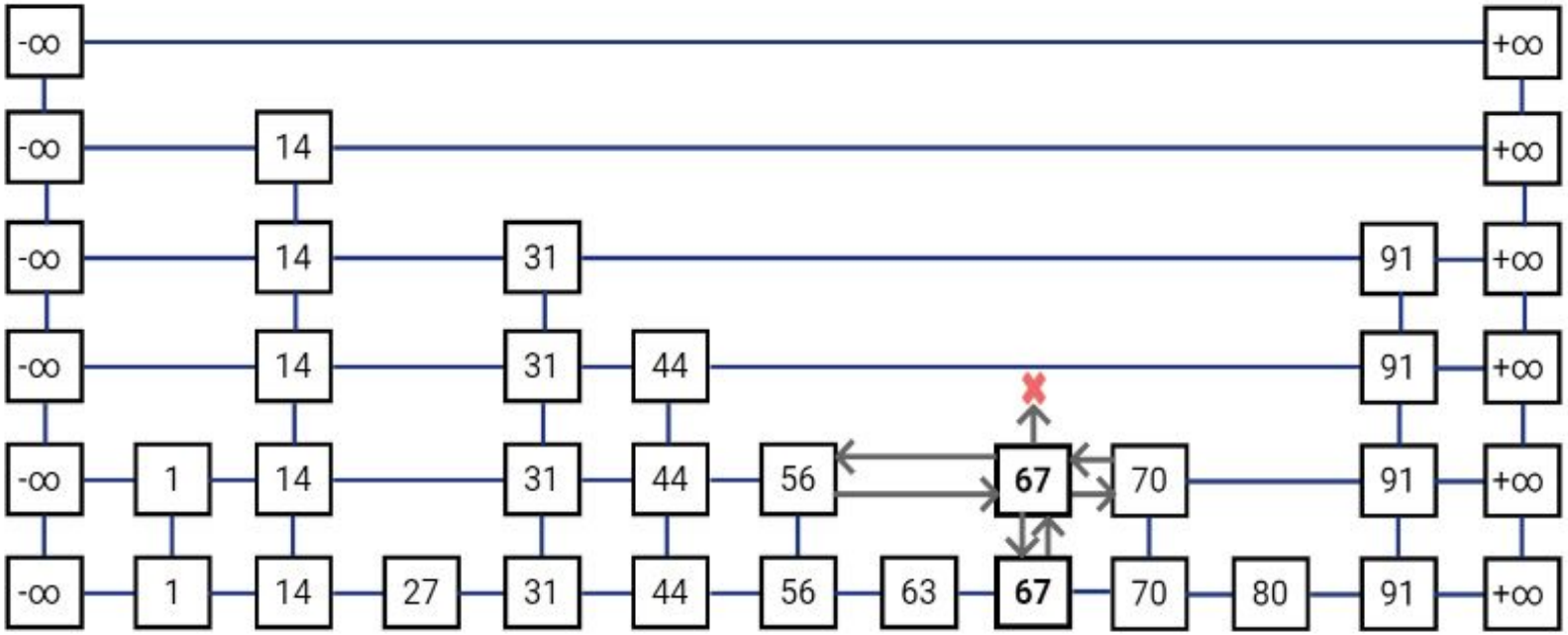
Insertion Operation:

- Insertion of a node in a skip list needs to be in such a manner that after insertion of the new node the list should still be in sorted order and there is a randomization method based on the probability to ensure that skewness will never happen and we will never lose the efficiency of operations on skip list.
- We first need to find a particular node in the bottom linked list, only then we will insert the new node at the appropriate location.

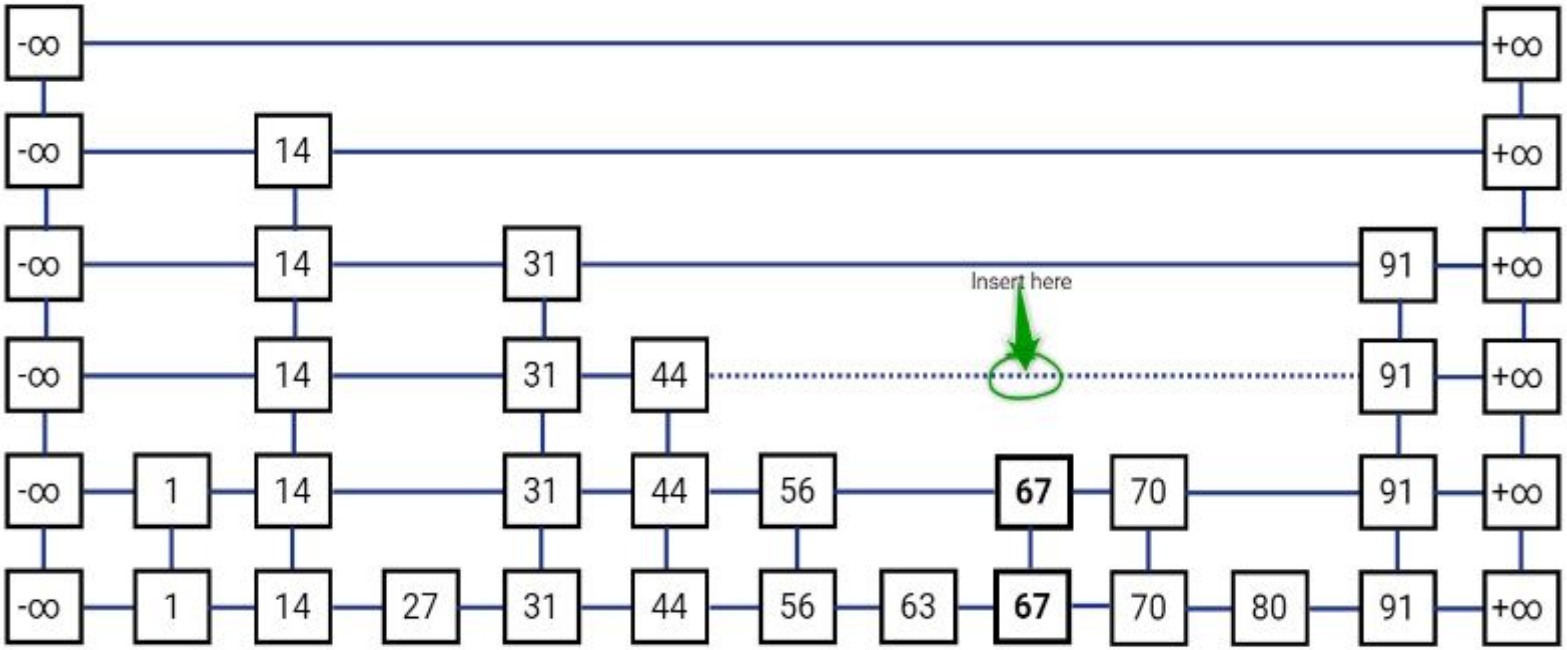

Inserting 67
-∞
+∞
14
31
91
1
44
56
70
27
63
80
Insert here


Inserting 67
-∞
+∞
-∞
14
+∞
-∞
14
31
91
+∞
-∞
14
31
44
91
+∞
-∞
1
14
31
44
56
70
91
+∞
-∞
1
14
27
31
44
56
63
67
70
80
91
+∞

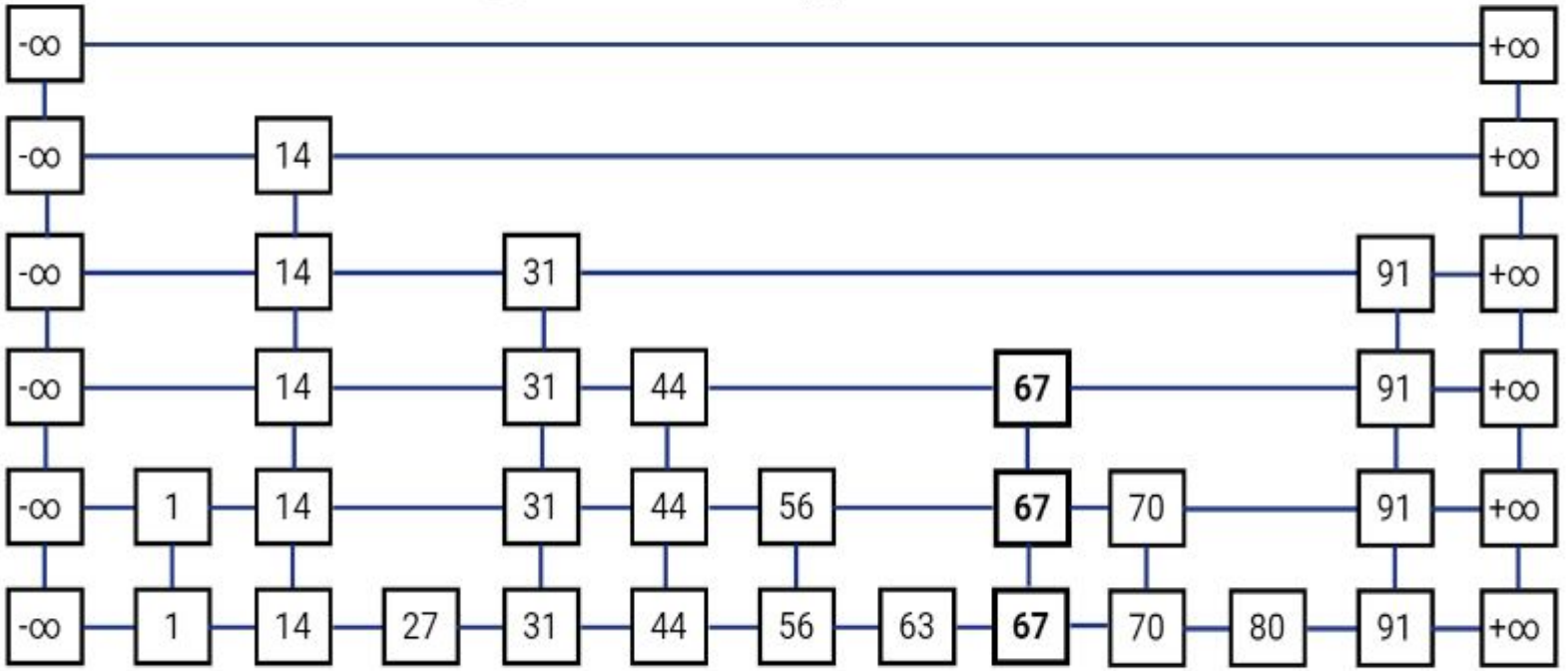

(Head)
-∞
+∞
14
31
91
44
Insert here
1
56
70
27
63
67
80



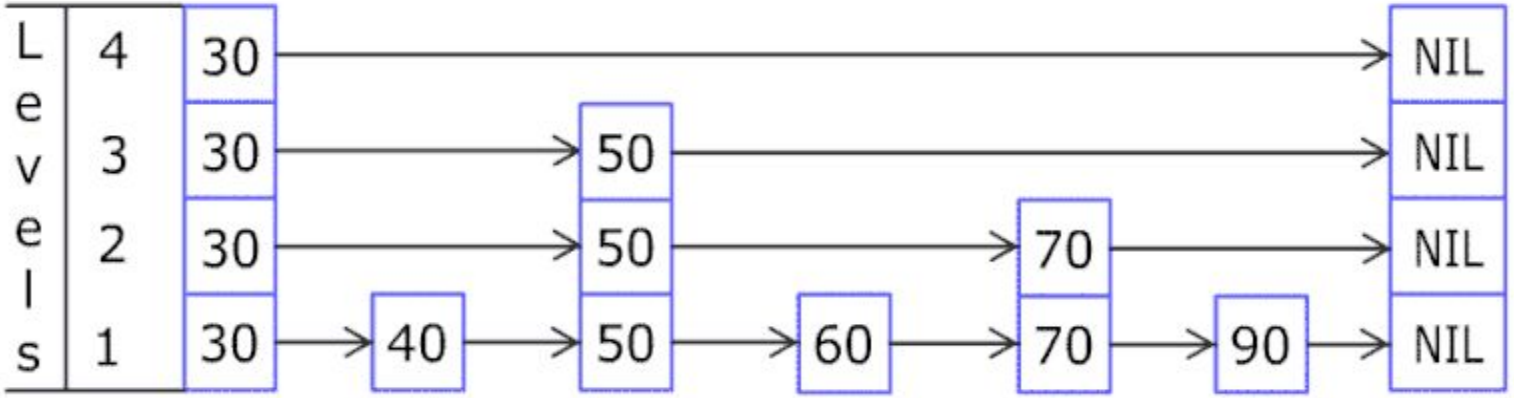
Let's toss the coin again.



Last toss gave us another Head, so let's toss the coin again.



We got a Tail this time, so no more insertions are required.

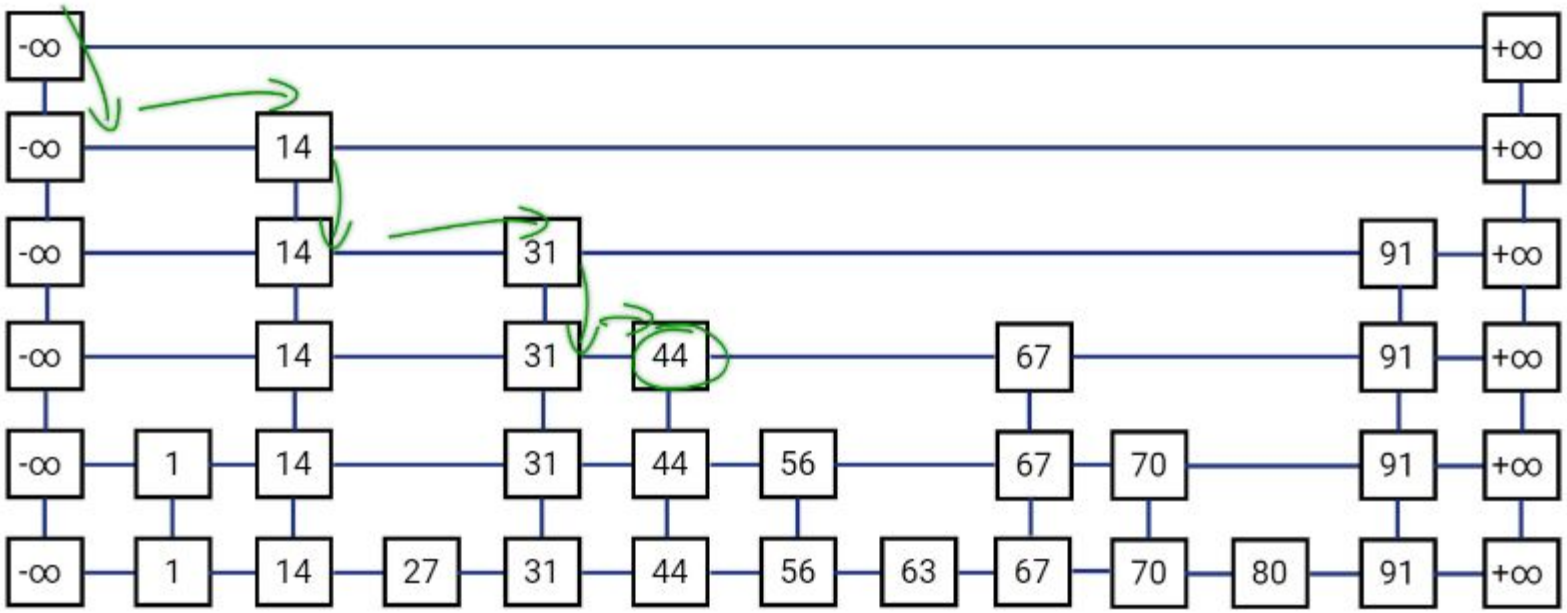

Levels
4
3
2
1
30
30
30
30
40
50
50
50
60
70
70
90
NIL
NIL
NIL
NIL

Deletion Operation -

For deletion operation, similar to insertion, we will first try to find the node to be deleted, and then we will do some pointers rearrangement to remove the node from the skip list, and finally, we can release the memory of that node .

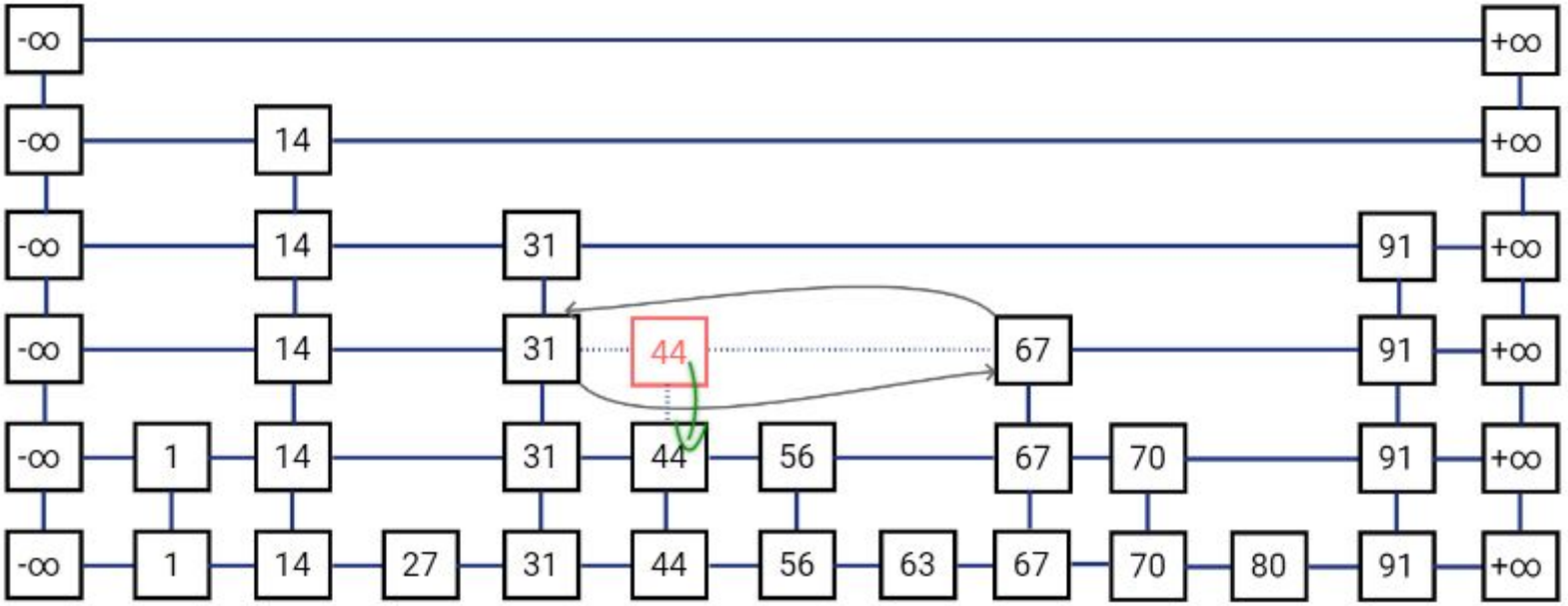
1. We will iterate through the layer till we find an element that is just greater than the node we are trying to delete.
2. Once we find such a node we will use its pointer to go down to the lower layer.
3. We will repeat steps 1 and 2 till we reach the bottom-most layer, once we reached the required node while performing step 1, do remove that node.

Deleting 44



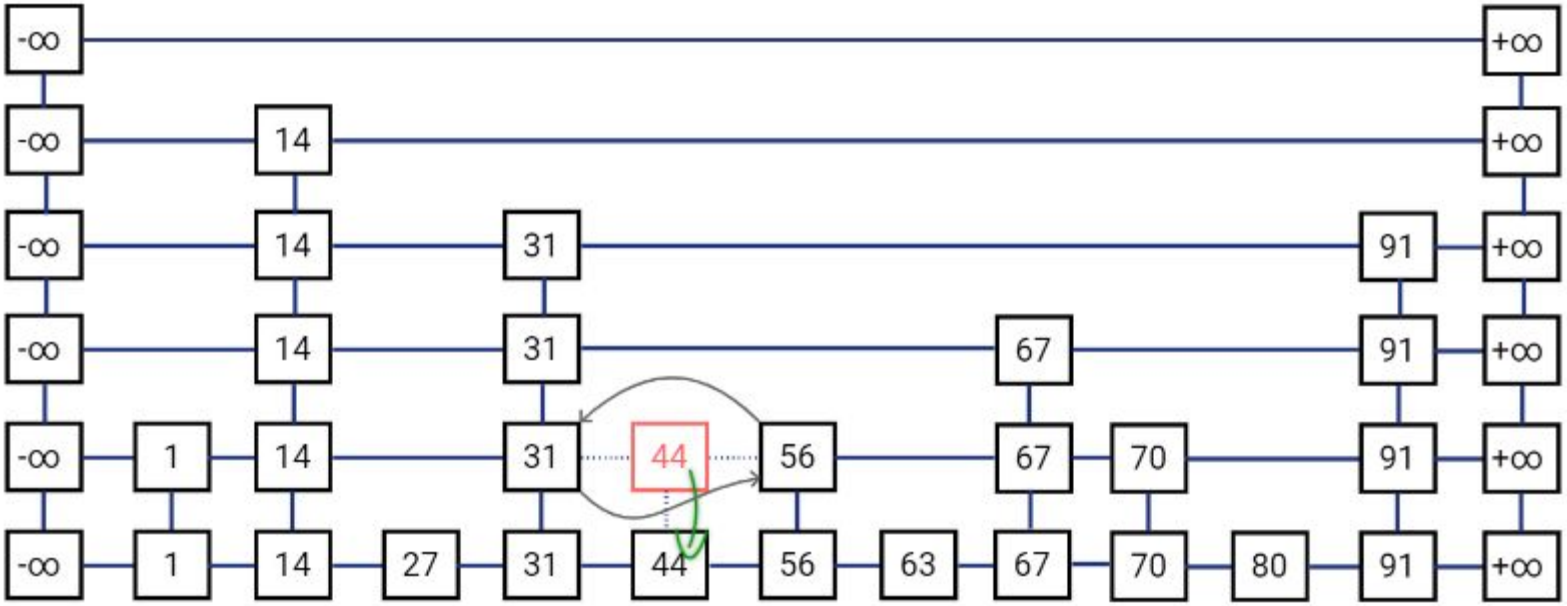
Release all references to this node. And move to the node below it.

Deleting 44



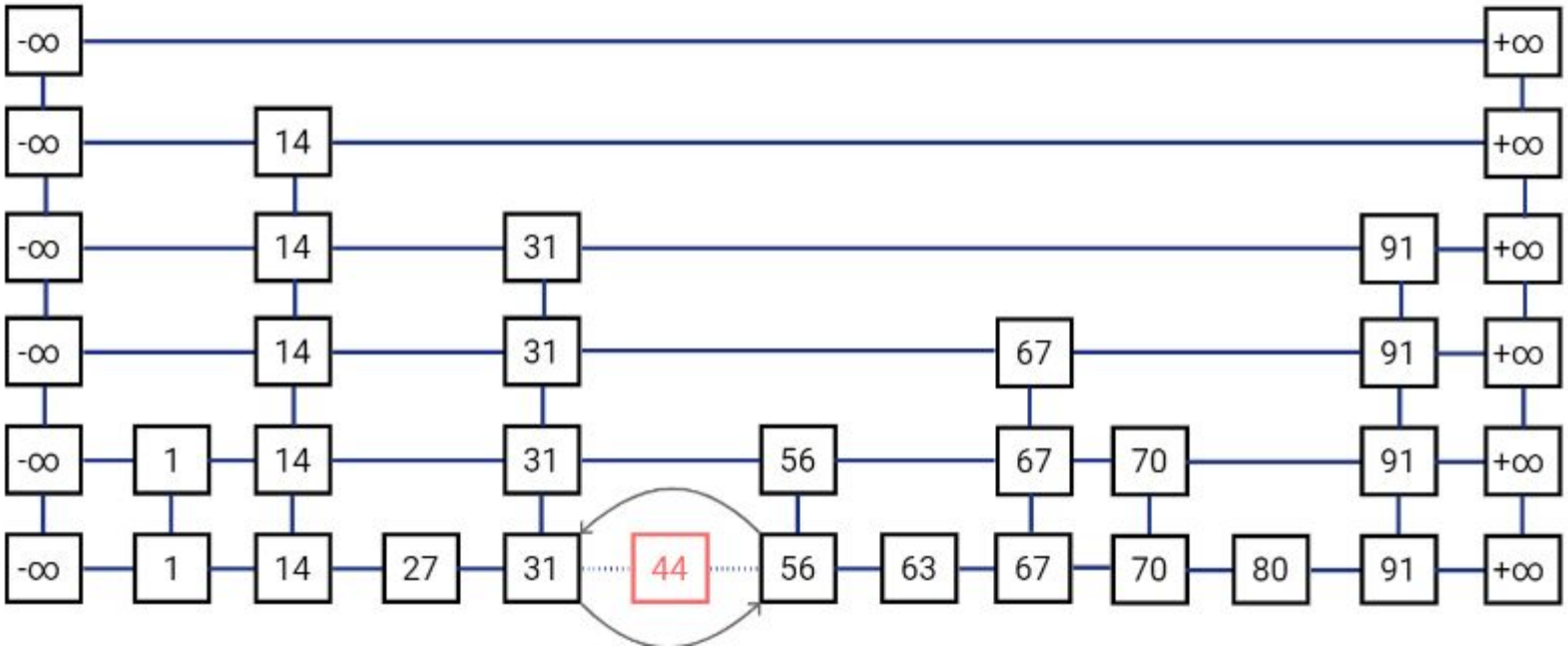
Remove all references to this node and release it. Move to the node below in the bottom list.

Deleting 44



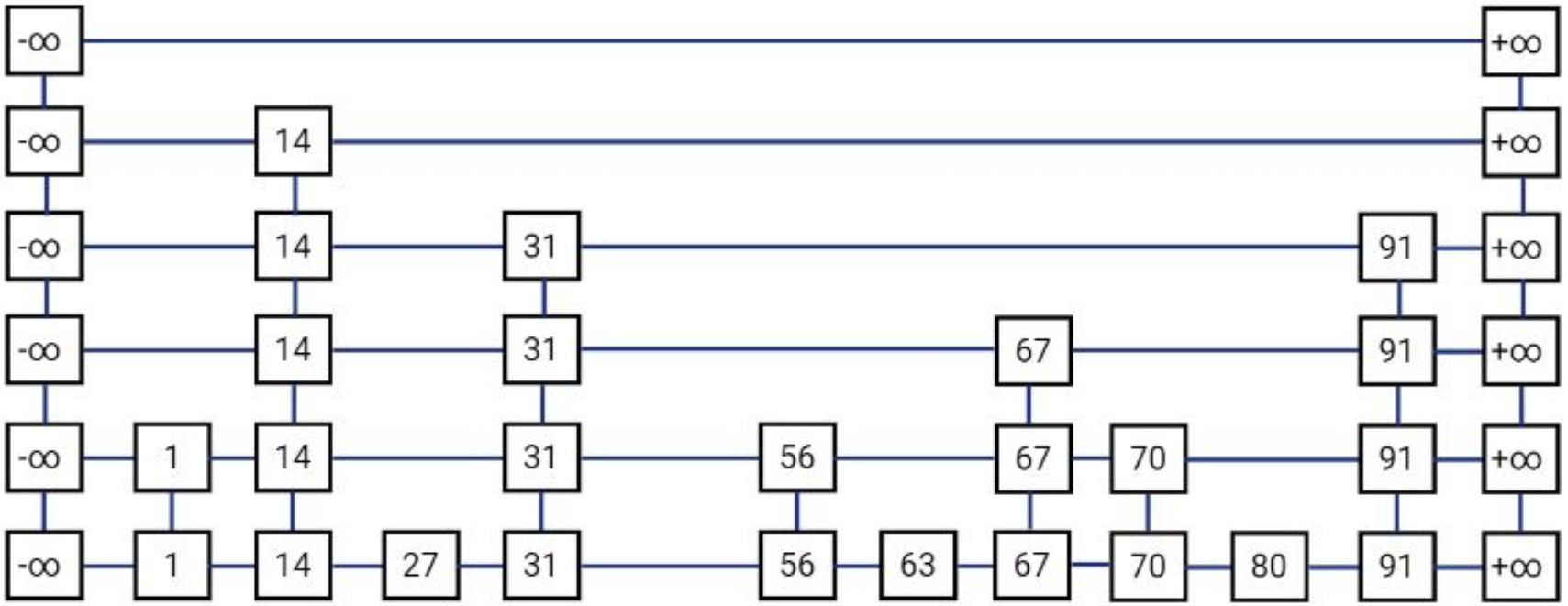
Release this node as well.

Deleting 44


-∞
+∞
-∞
14
+∞
-∞
14
31
91
+∞
-∞
14
31
67
91
+∞
-∞
1
14
31
56
67
70
91
+∞
-∞
1
14
27
31
44
56
63
67
70
80
91
+∞

This is how our Skip List looks like after deletion.

Deleting 44



Insertion Operation -

Before we start inserting the elements in the skip list we need to decide the nodes level. Each element in the list is represented by a node, the level of the node is chosen randomly while insertion in the list. Level does not depend on the number of elements in the node. The level for node is decided by the following algorithm –

```
randomLevel()
lvl := 1
//random() that returns a random value in [0...1)
while random() < p and lvl < MaxLevel do
lvl := lvl + 1
return lvl
```

MaxLevel is the upper bound on number of levels in the skip list. It can be determined as – $\mathbf{L(N) = \log_{p/2}(N)}$. Above algorithm assure that random level will never be greater than MaxLevel. Here p is the fraction of the nodes with level i pointers also having level i+1 pointers and N is the number of nodes in the list.

Pseudocode For Insertion

```
Insert(list, searchKey)
local update[0...MaxLevel+1]
x := list -> header
for i := list -> level down to 0 do
       while x -> forward[i] -> key  forward[i]
update[i] := x
x := x -> forward[0]
lvl := randomLevel()
if lvl > list -> level then
for i := list -> level + 1 to lvl do
      update[i] := list -> header
      list -> level := lvl
x := makeNode(lvl, searchKey, value)
for i := 0 to level do
      x -> forward[i] := update[i] -> forward[i]
update[i] -> forward[i] := x
```

Searching Operation - Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if –

1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] and move one level down and continue our search.

At the lowest level (0), if the element next to the rightmost element (update[0]) has key equal to the search key, then we have found key otherwise failure.

Pseudocode For Searching:

```
Search(list, searchKey)
        x := list -> header
        -- loop invariant: x -> key  level downto 0 do
             while x -> forward[i] -> key  forward[i]
        x := x -> forward[0]
       if x -> key = searchKey then return x -> value
       else return failure
```

Deletion Operation - Deletion of an element k is preceded by locating element in the Skip list using above mentioned search algorithm. Once the element is located, rearrangement of pointers is done to remove element from list just like we do in singly linked list. We start from lowest level and do rearrangement until element next to update[i] is not k. After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list.

Pseudocode For Deletion

```
Delete(list, searchKey)
local update[0..MaxLevel+1]
x := list -> header
for i := list -> level down to 0 do
   while x -> forward[i] -> key  forward[i]
      update[i] := x
x := x -> forward[0]
if x -> key = searchKey then
   for i := 0 to list -> level do
        if update[i] -> forward[i] ≠ x then break
        update[i] -> forward[i] := x -> forward[i]
   free(x)
   while list -> level > 0 and list -> header -> forward[list -> level] = NIL do
        list -> level := list -> level – 1
```