RV UNIVERSITY

प्रज्ञा: ★ धीरा:

Go, change the world

an initiative of RV EDUCATIONAL INSTITUTIONS

Unit-3

Content:

- Searching,Sorting and Hashing
  - Linear Search
  - Binary search
  - Bubble sort
  - Selection sort
  - Insertion Sort
  - Hashing-Hash tables
  - Hash functions
  - Strategies to avoid and resolve collisions.

Searching

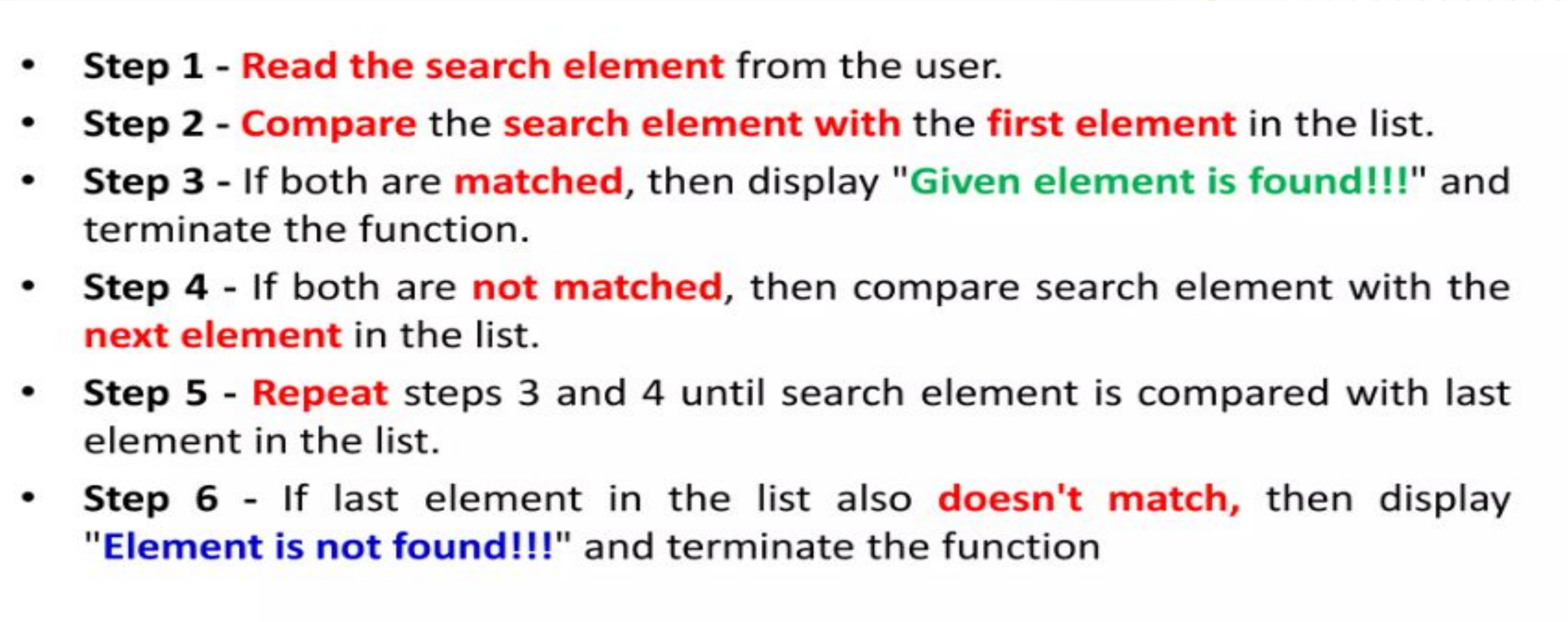
- Search is process of finding a value in a list of values
- In other words, searching is the process of locating given value position in a list of values.

- Types of searching
  - Linear search
  - Binary Search
  - Interpolation Search
  - Sublist Search
  - Exponential Search
  - Jump Search
  - Fibonacci Search and etc.

Linear Search

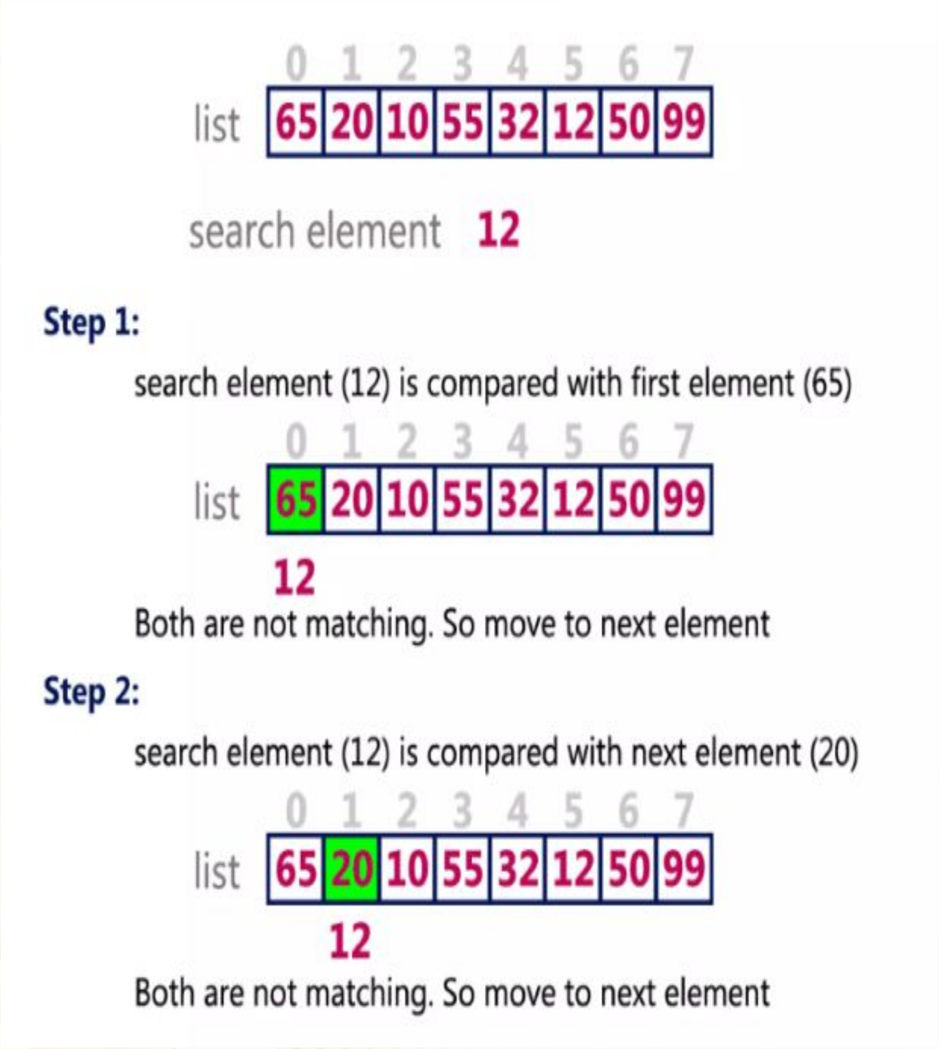
- Linear search algorithm finds a given element in a list of elements with O(n) time complexity where n is the total number of elements in a list.
- Search process starts comparing search element with the first element in the list.
- If both are matched then result is element found otherwise search element is compared with the next element in the list.
- Repeat the same until search element is compared with the last element in the list, if that last element also doesn’t match, then the result is **“Element is not found in the list”.**
- That means, the search element is compared with element by element in the list.

Linear Search Algorithm



- **Step 1** - **Read the search element** from the user.
- **Step 2** - **Compare** the **search element with** the **first element** in the list.
- **Step 3** - If both are **matched**, then display "**Given element is found!!!**" and terminate the function.
- **Step 4** - If both are **not matched**, then compare search element with the **next element** in the list.
- **Step 5** - **Repeat** steps 3 and 4 until search element is compared with last element in the list.
- **Step 6** - If last element in the list also **doesn't match,** then display "**Element is not found!!!**" and terminate the function

Linear Search Example



	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element 12

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	**65**	20	10	55	32	12	50	99
	12							

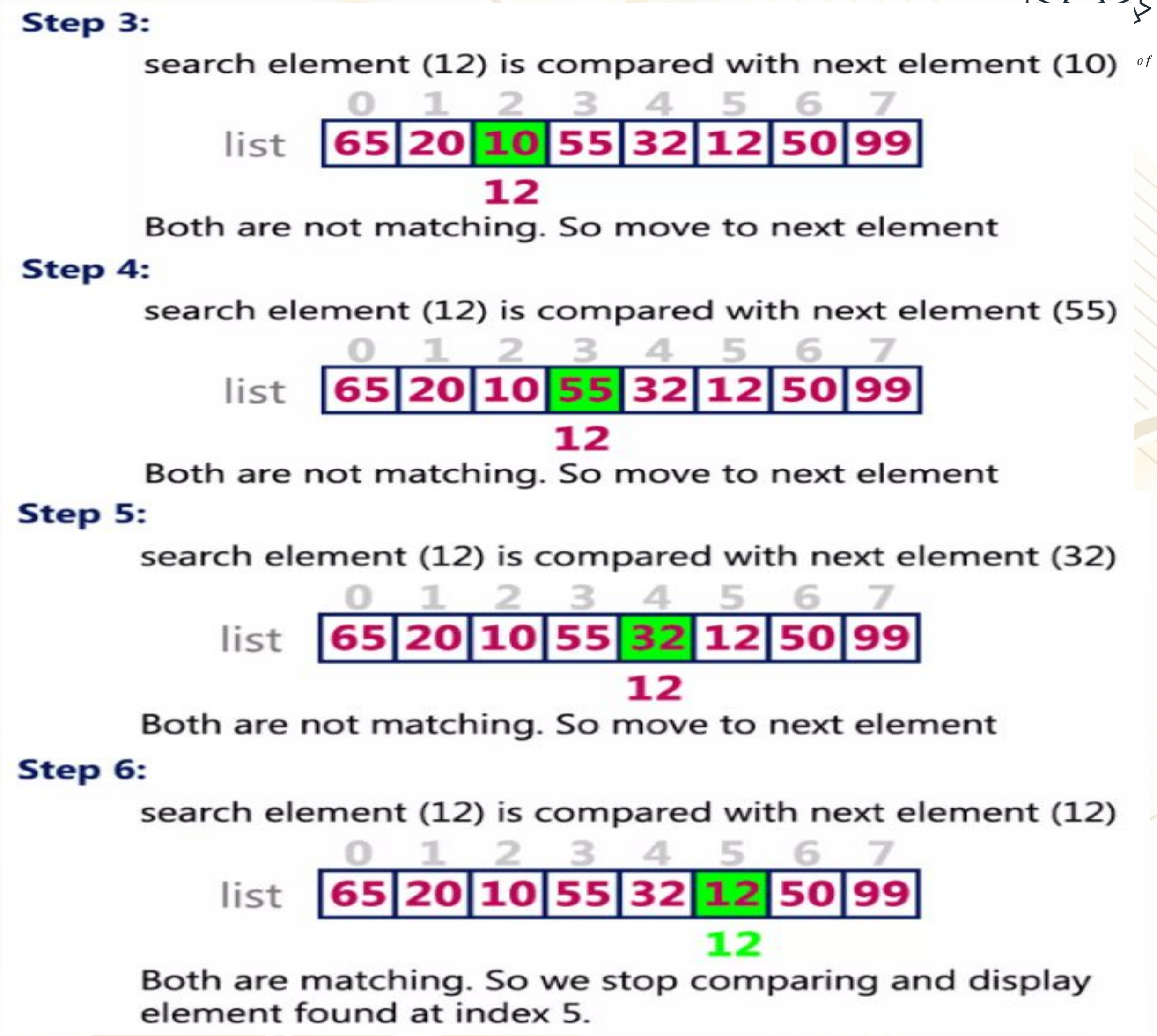
Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	**20**	10	55	32	12	50	99
		12						

Both are not matching. So move to next element



Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	**10**	55	32	12	50	99
			12					

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	**55**	32	12	50	99
				12				

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	**32**	12	50	99
					12			

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	**12**	50	99
						12		

Both are matching. So we stop comparing and display element found at index 5.

Program

```
#include <stdio.h>

int main() {

    int size, target, i;

    int found = -1;  // Flag to indicate if the element is found (-1 means not found)

    // Reading the size of the array from the user

    printf("Enter the size of the array: ");

    scanf("%d", &size);

    int arr[size];  // Declare the array with the given size
// Reading array elements from the user
    printf("Enter %d elements for the array: \n", size);
    for (i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
// Linear Search to find the target
    for (i = 0; i < size; i++) {
        if (arr[i] == target) {
            found = i;  // Target found at index i
            break;
        }   }
// Output the result of the search
    if (found != -1) {
        printf("Element %d found at index %d\n", target, found);
    } else {
        printf("Element %d not found in the array\n", target);
    }
    return 0;
}
```

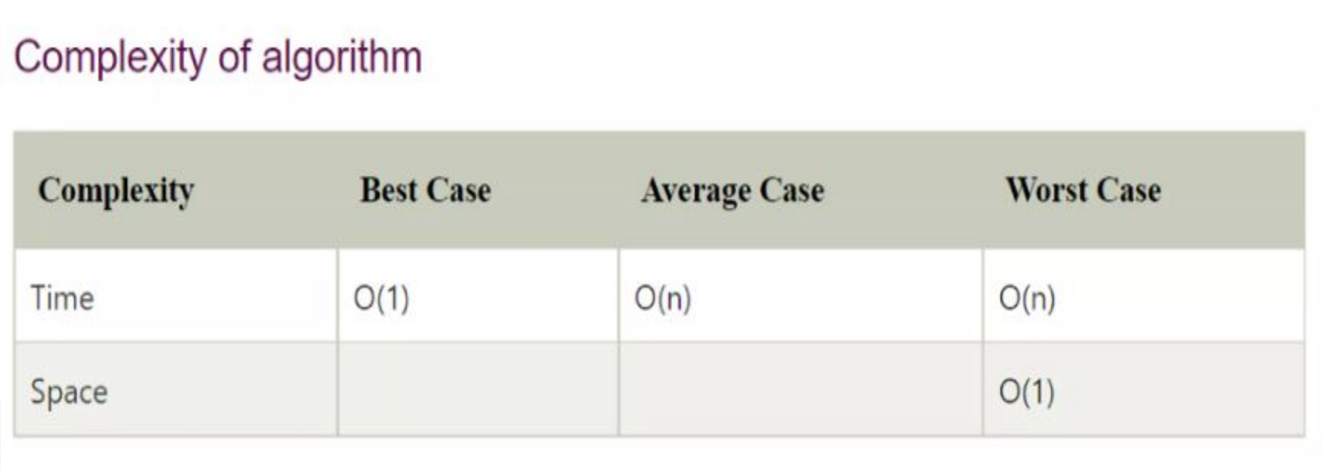
Output

Output-1

```
Enter the size of the array: 6

Enter 6 elements for the array:

2 3 4 5 6 7

Enter the element to search: 4

Element 4 found at index 2
```

Output-2

```
Enter the size of the array: 6
Enter 6 elements for the array:
1 2 3 4 5 6
Enter the element to search: 8
Element 8 not found in the array
```



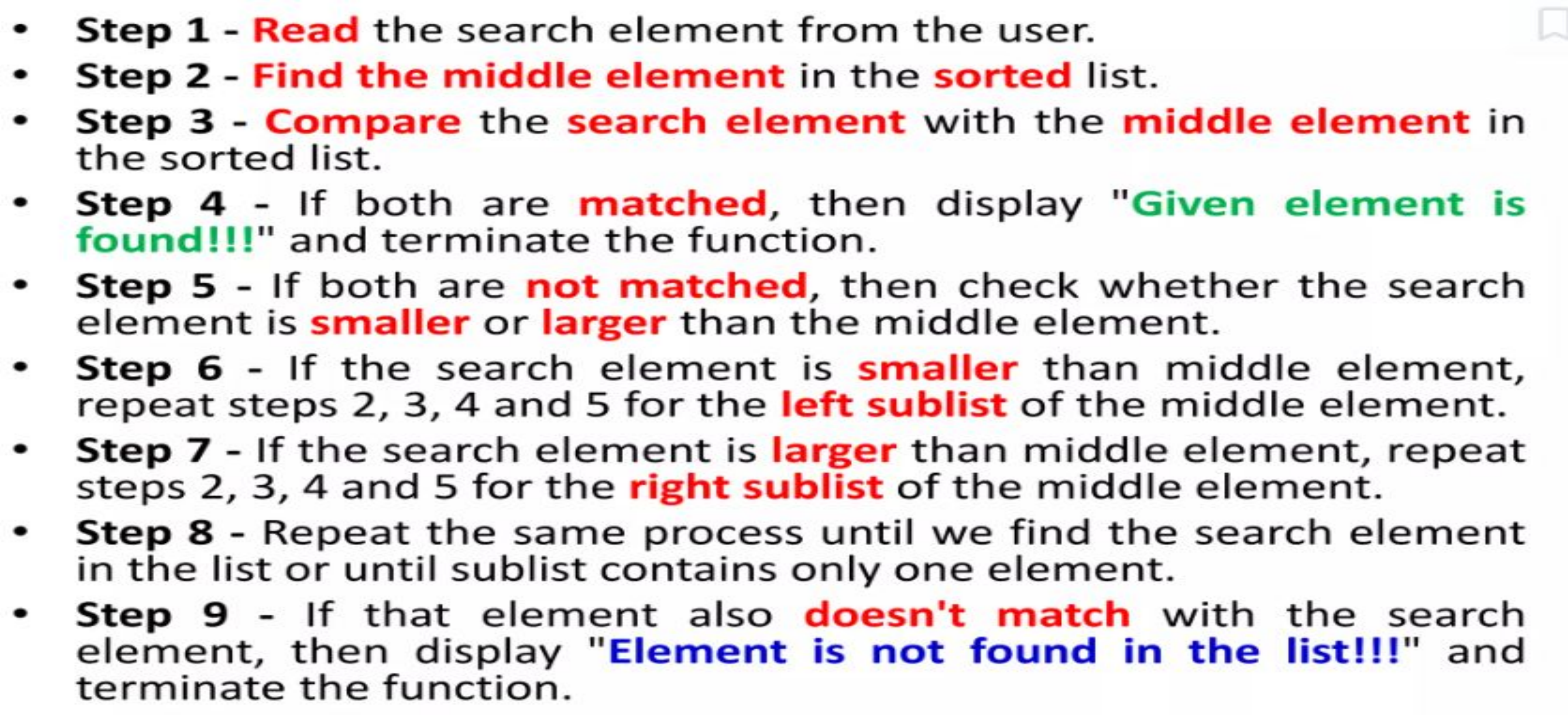
Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

Binary Search

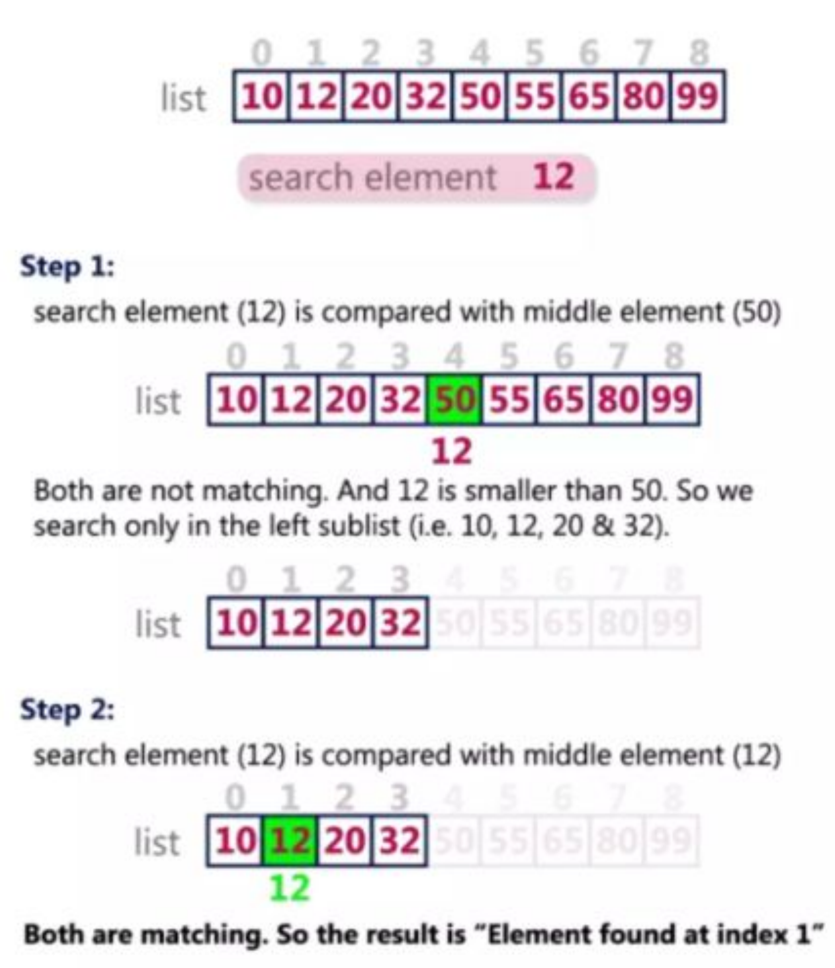
- Finds given element in a list of elements with O(logn) time complexity where n is the total number of elements in the list.
- Used with only sorted list of elements.
- Means used only with list of elements that are already arranged in an order.
- Can not be used for list of elements arranged in random order.
- Search process starts comparing the search element with the middle element in the list.

Binary Search Algorithm



- **Step 1** - **Read** the search element from the user.
- **Step 2** - **Find the middle element** in the **sorted** list.
- **Step 3** - **Compare** the **search element** with the **middle element** in the sorted list.
- **Step 4** - If both are **matched**, then display "**Given element is found!!!**" and terminate the function.
- **Step 5** - If both are **not matched**, then check whether the search element is **smaller** or **larger** than the middle element.
- **Step 6** - If the search element is **smaller** than middle element, repeat steps 2, 3, 4 and 5 for the **left sublist** of the middle element.
- **Step 7** - If the search element is **larger** than middle element, repeat steps 2, 3, 4 and 5 for the **right sublist** of the middle element.
- **Step 8** - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9** - If that element also **doesn't match** with the search element, then display "**Element is not found in the list!!!**" and terminate the function.

Example



list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

search element 12

Step 1:

search element (12) is compared with middle element (50)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 2:

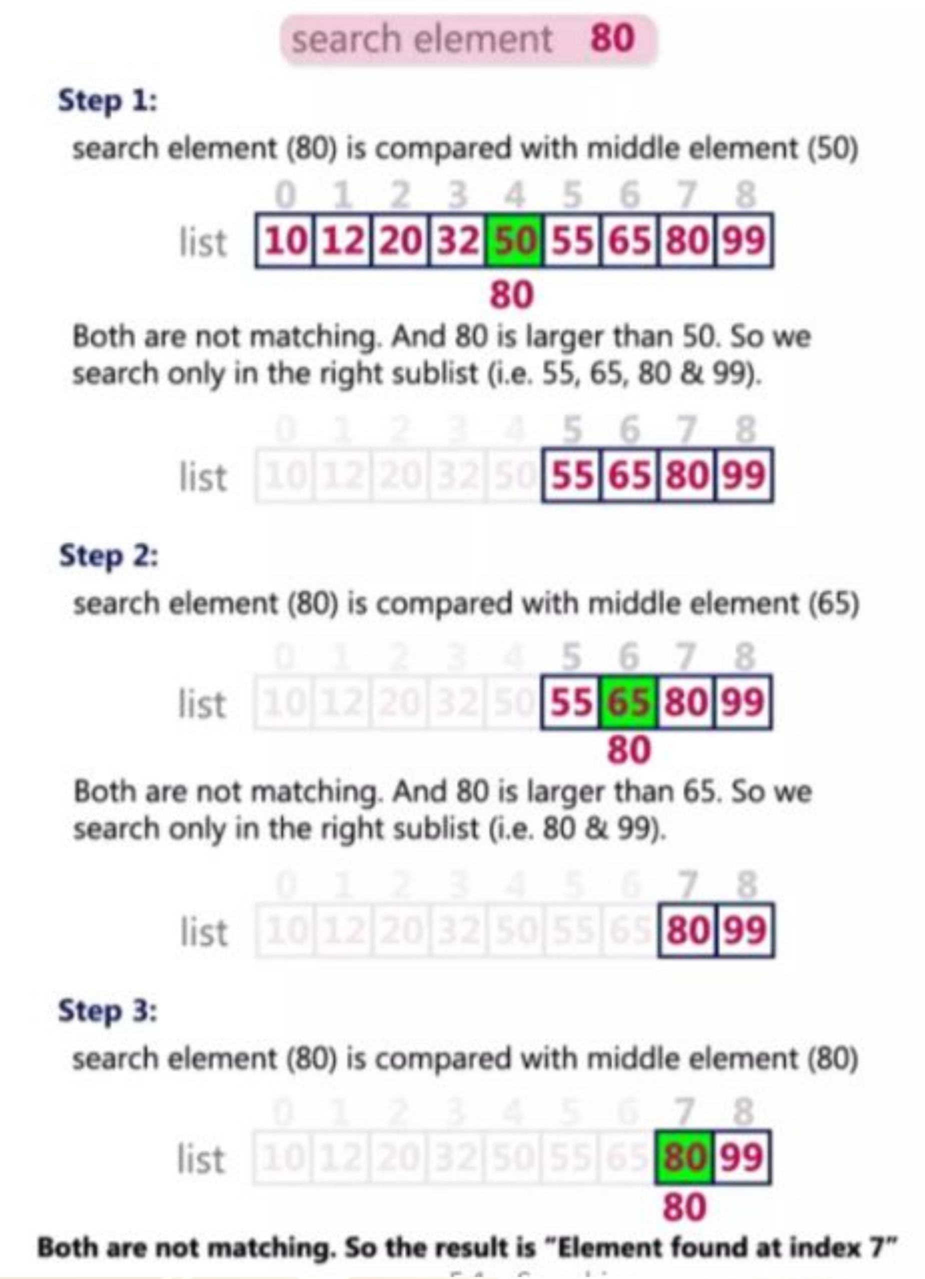
search element (12) is compared with middle element (12)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

12

Both are matching. So the result is "Element found at index 1"



search element 80

Step 1:

search element (80) is compared with middle element (50)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. So the result is "Element found at index 7"

Program

```
#include <stdio.h>

 int main() {

   int size, target, low, high, mid;

   // Reading the size of the array from the user

   printf("Enter the size of the array: ");

   scanf("%d", &size);

   int arr[size];  // Declare the array with the given size

   // Reading array elements from the user

   printf("Enter %d sorted elements for the array (in
ascending order): \n", size);

for (int i = 0; i < size; i++) {

     scanf("%d", &arr[i]);

  }
```

```
// Reading the target element to search for

   printf("Enter the element to search: ");

   scanf("%d", &target);

   // Binary Search (Iterative)

   low = 0;   high = size - 1;

   int found = -1;  // Flag to indicate if the
element is found (-1 means not found)

   while (low <= high) {

      mid = (low + high) / 2;  // Find the middle
index

while (low <= high) {

      mid = (low + high) / 2;  // Find the middle
index

if (arr[mid] == target) {

      found = mid;  // Target found at index mid

break;

} else if (arr[mid] < target) {
```

```
low = mid + 1;  // Target is in the right half

      } else {

         high = mid - 1;  // Target is in the left
half

      }

   }

   // Output the result of the search

   if (found != -1) {

      printf("Element %d found at index %d\n",
target, found);

   } else {

      printf("Element %d not found in the
array\n", target);

   }

   return 0;

}
```

Output

Output:

```
Enter the size of the array: 5
Enter 5 sorted elements for the
array (in ascending order):
1 2 3 4 5
Enter the element to search: 5
Element 5 found at index 4
```

Output:

```
Enter the size of the array: 6
Enter 6 sorted elements for the array (in ascending order):
1 2 3 4 5 6
Enter the element to search: 7
Element 7 not found in the array
```

Sorting

- Arrange of data in a preferred order.
- By sorting a data, it is easier to search through it quickly and easily.
- Simplex example of sorting is dictionary.

Bubble Sort

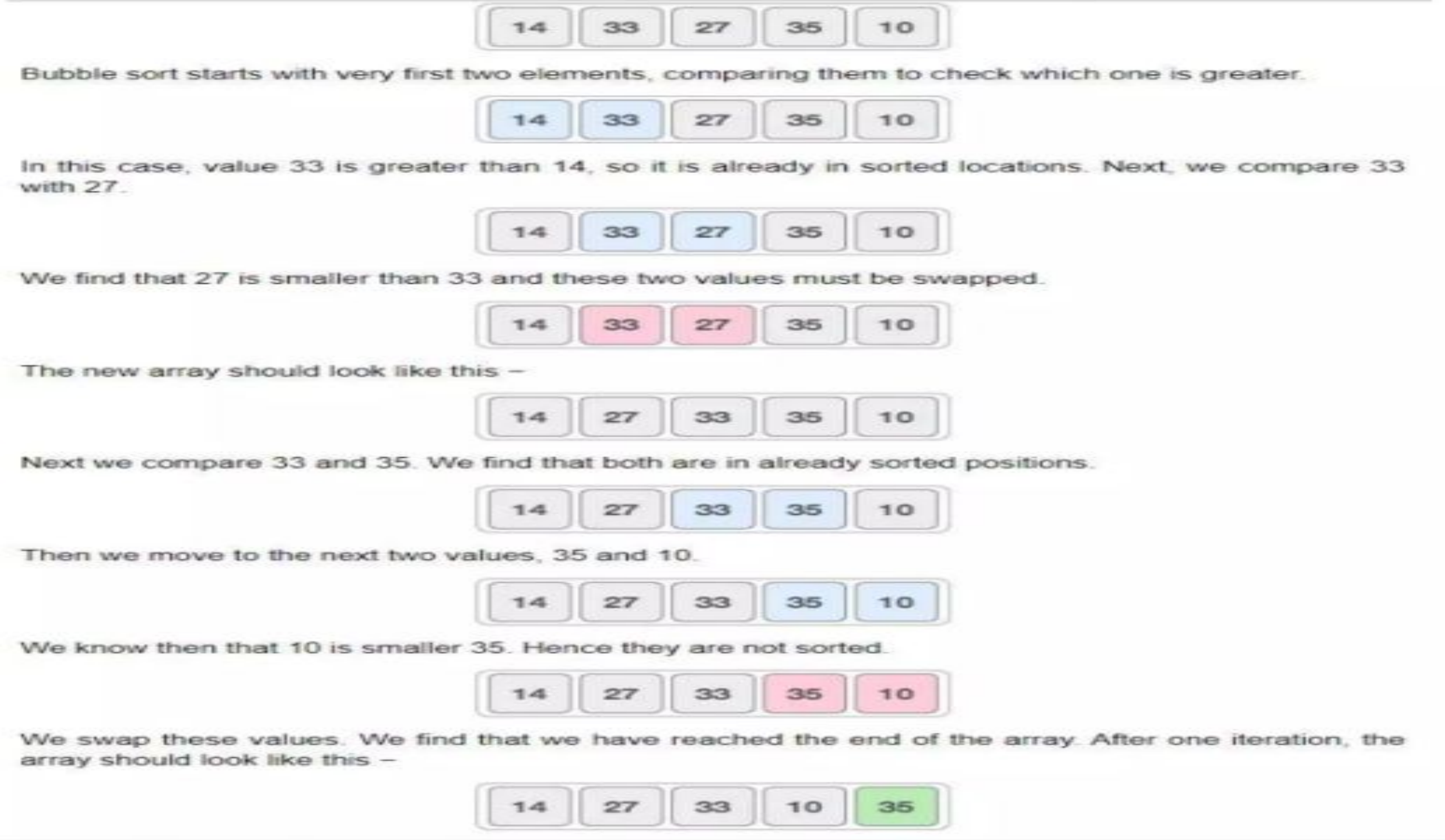
-
  - Its simple sorting algorithm
  - Comparison based algorithm in which each pair of adjustment elements is compared and the elements are swapped if they are not in order.
  - Not suitable for large datasets as its average and worst case complexity of O(n^2) where n is the number of items.

Bubble Sort Algorithm

- We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
Begin BubbleSort(list)
    for all elements of list
        if list[i]>list[i+1]
            swap(list[i], list[i+1])
        end if
    end for
    return list
End BubbleSort
```

Bubble Sort Example



14	33	27	35	10

Bubble sort starts with very first two elements, comparing them to check which one is greater.

14	33	27	35	10

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

14	33	27	35	10

We find that 27 is smaller than 33 and these two values must be swapped.

14	33	27	35	10

The new array should look like this –

14	27	33	35	10

Next we compare 33 and 35. We find that both are in already sorted positions.

14	27	33	35	10

Then we move to the next two values, 35 and 10.

14	27	33	35	10

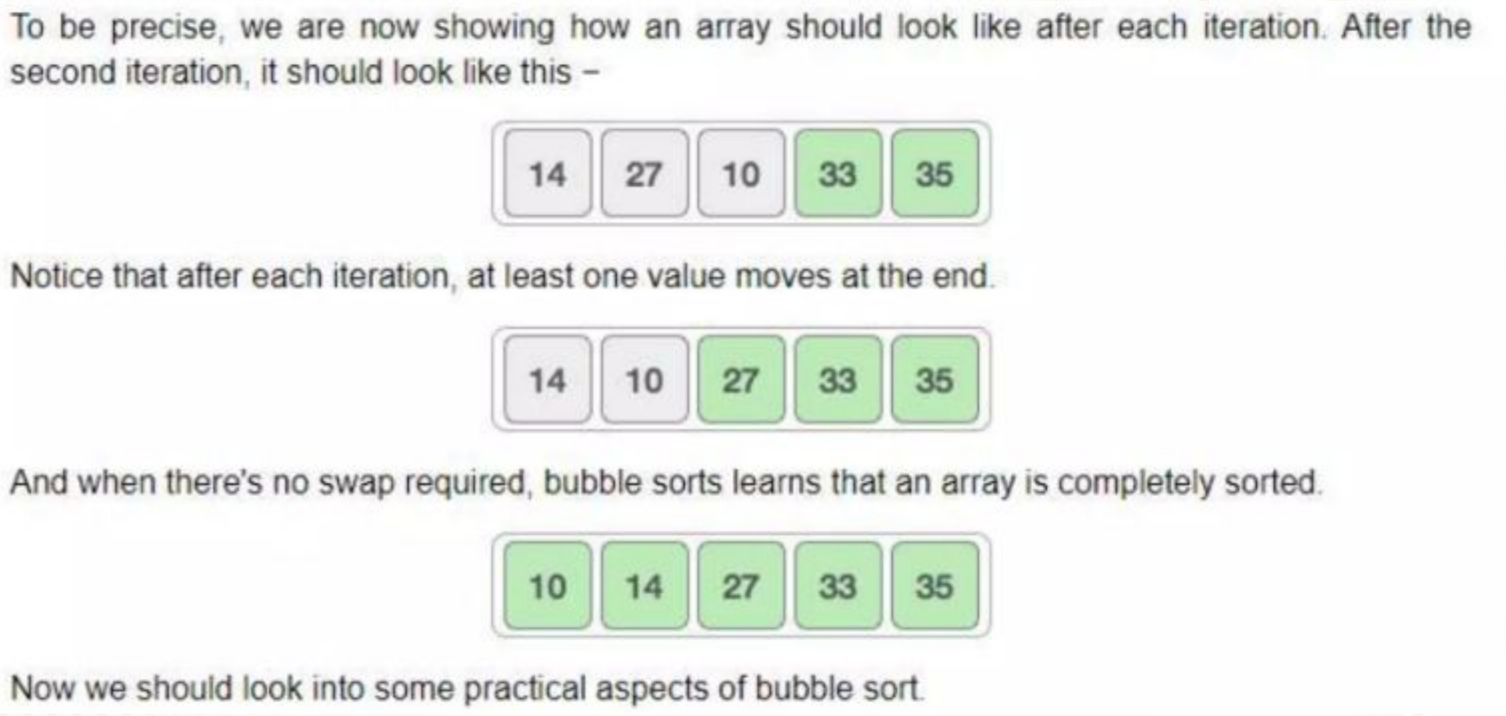
We know then that 10 is smaller 35. Hence they are not sorted.

14	27	33	35	10

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

14	27	33	10	35

Bubble Sort Example cont…



To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

14	27	10	33	35

Notice that after each iteration, at least one value moves at the end.

14	10	27	33	35

And when there's no swap required, bubble sorts learns that an array is completely sorted.

10	14	27	33	35

Now we should look into some practical aspects of bubble sort.

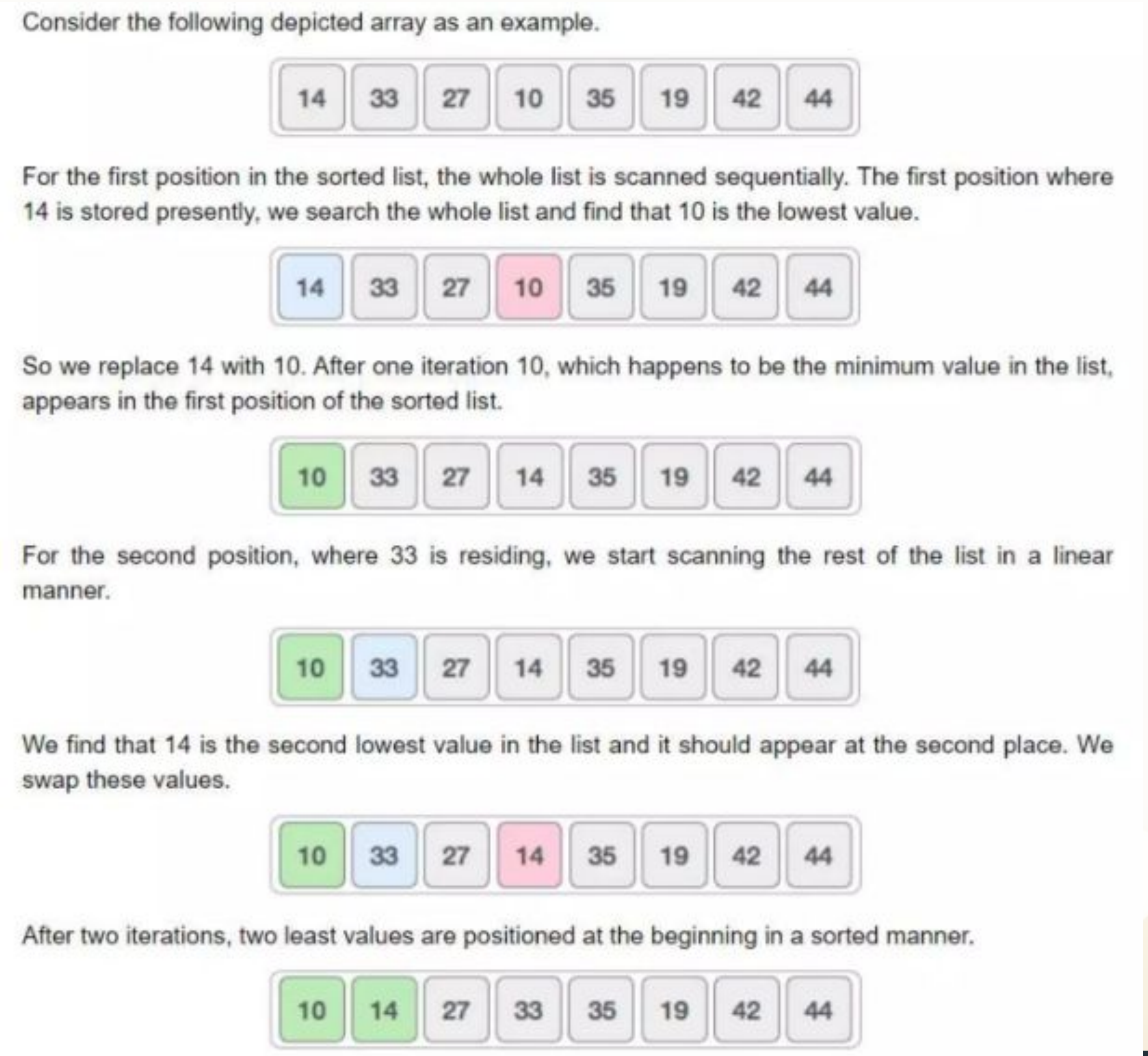
Selection Sort

- - Its an in-place comparison-based algorithm in which the list is divided into 2 parts, the sorted part at the left end, and the unsorted part at the right end.
  - Initially the sorted part is empty and the unsorted is the entire list.
  - The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
  - This process continues moving unsorted array boundary by one element to the right.
  - This algorithm is not suitable for large dataset as its average and worst case complexities are of O(n^2), where n is the number of items.

Selection Sort Algorithm

- Step1-Set MIN to location 0
- Step2-Search the minimum element in the list
- Step3-Swap with value at location MIN
- Step4-Increment MIN to point to next element.
- Step5-Repeat until list is sorted

Example



Consider the following depicted array as an example.

14	33	27	10	35	19	42	44

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

14	33	27	10	35	19	42	44

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

10	33	27	14	35	19	42	44

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

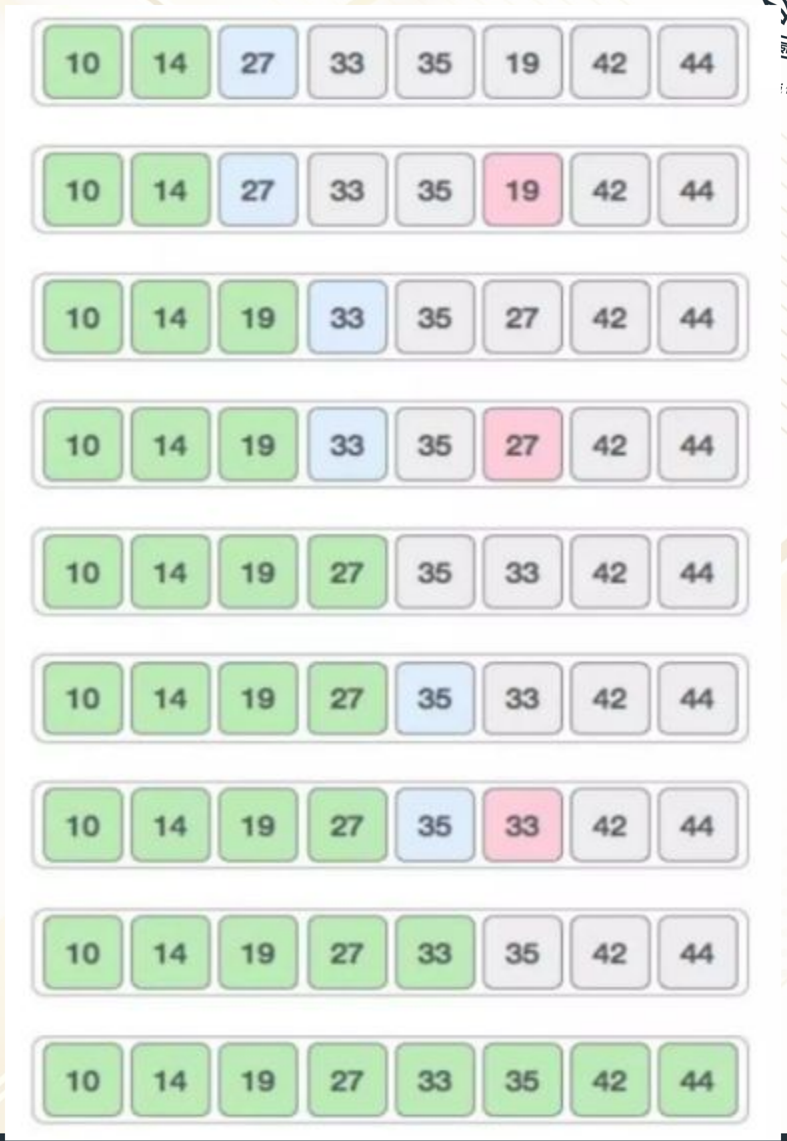
10	33	27	14	35	19	42	44

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

10	33	27	14	35	19	42	44

After two iterations, two least values are positioned at the beginning in a sorted manner.

10	14	27	33	35	19	42	44



10	14	27	33	35	19	42	44
10	14	27	33	35	19	42	44
10	14	19	33	35	27	42	44
10	14	19	33	35	27	42	44
10	14	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	35	33	42	44
10	14	19	27	33	35	42	44
10	14	19	27	33	35	42	44

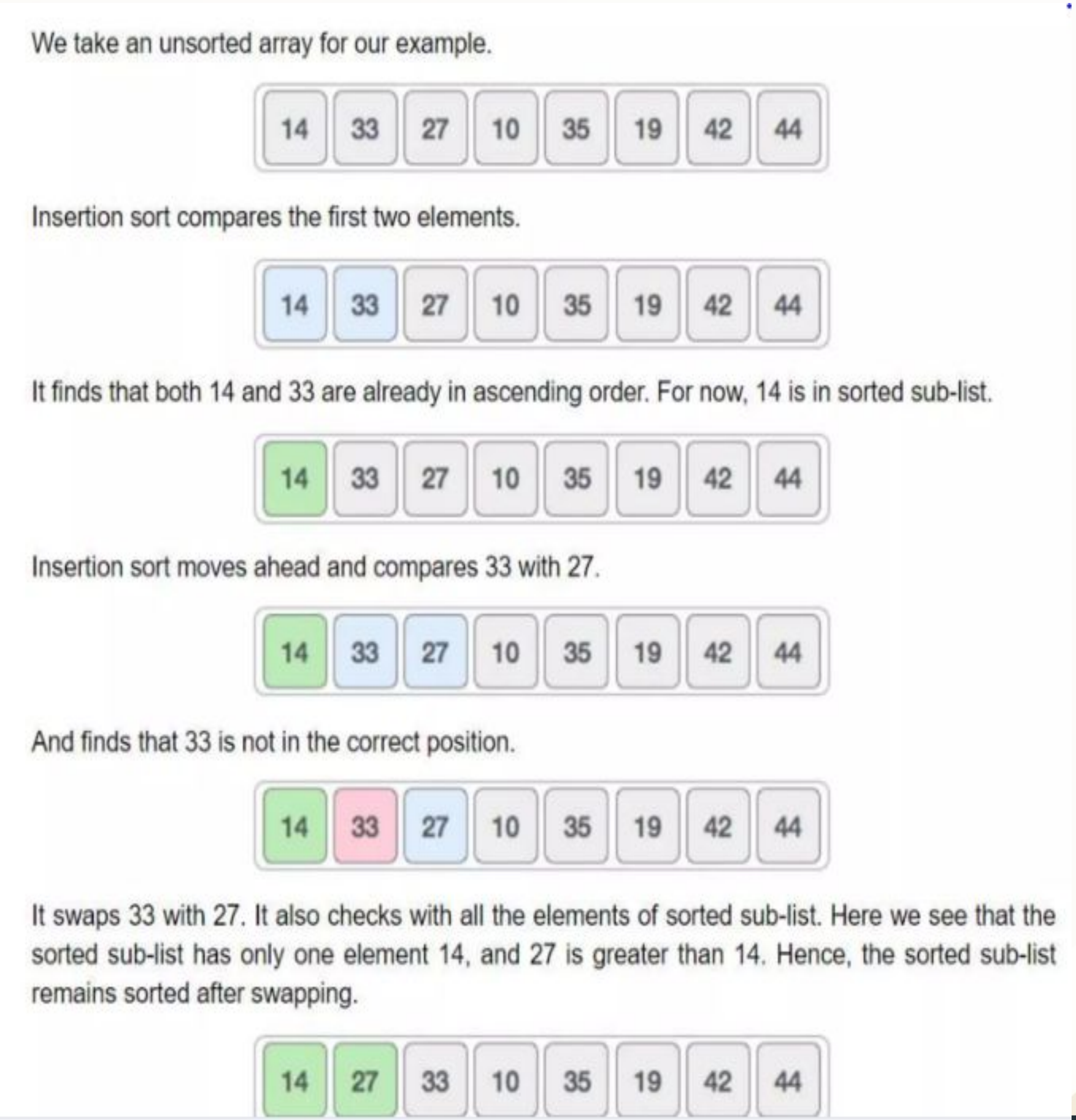
Insertion Sort

-
  - Its simple sorting algorithm that works similar to the way you sort playing cards in your hands.
  - The array is virtually split into a sorted and unsorted part
  - Values from the unsorted part are picked and placed at the correct position in the sorted part.
  - Its an in-place comparison-based sorting algorithm.
  - Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted.
  - An element which is to inserted in the sorted sub-list, has to find its appropriate place and then it has to be inserted there, hence the name insertion sort.
  - The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list(in the same array).

Insertion Sort Algorithm

- To sort an array of size n in descending order:
  - Iterate from arr[1] to arr[n] over the array
  - Compare the current element (key) do its predecessors.
  - If the key element is smaller than its predecessors, compare it to the elements before. Move the greater elements one position up to mark space for the swapped element.

Example:



We take an unsorted array for our example.

14 33 27 10 35 19 42 44

Insertion sort compares the first two elements.

14 33 27 10 35 19 42 44

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

14 33 27 10 35 19 42 44

Insertion sort moves ahead and compares 33 with 27.

14 33 27 10 35 19 42 44

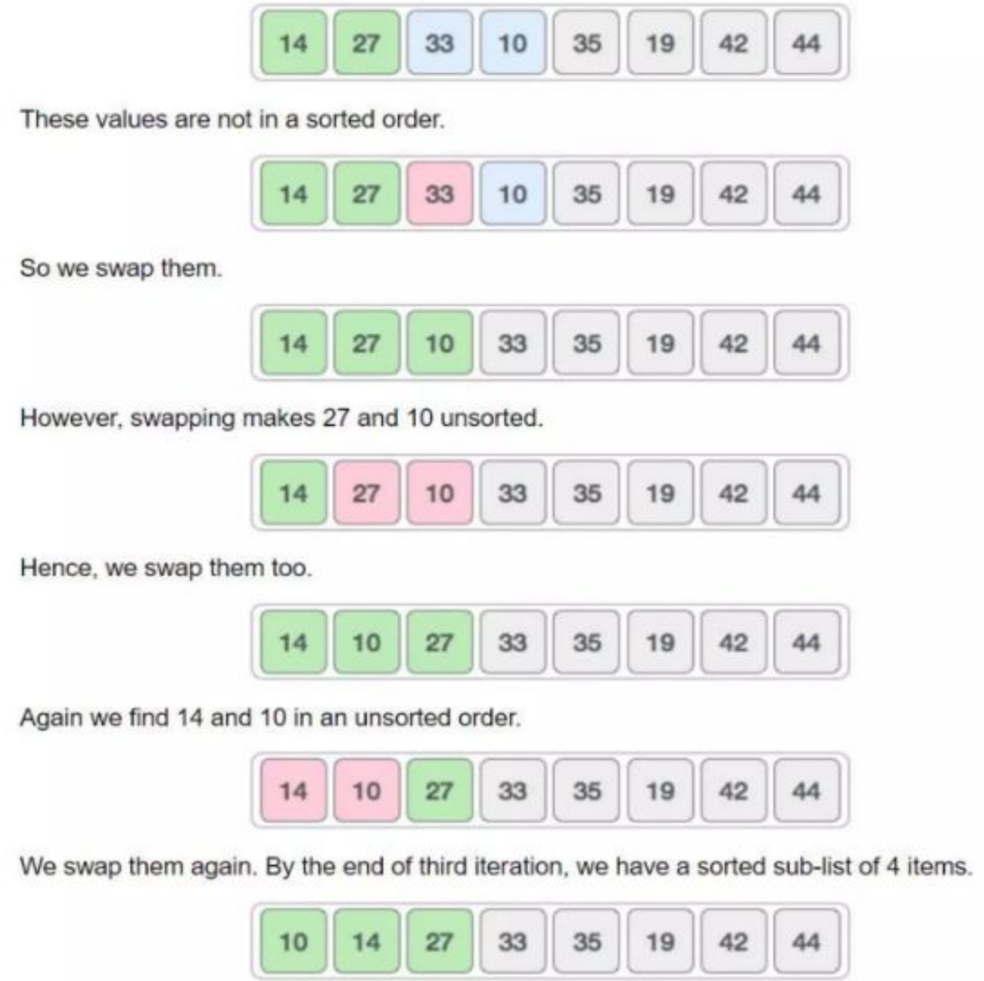
And finds that 33 is not in the correct position.

14 33 27 10 35 19 42 44

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

14 27 33 10 35 19 42 44

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



14 27 33 10 35 19 42 44

These values are not in a sorted order.

14 27 33 10 35 19 42 44

So we swap them.

14 27 10 33 35 19 42 44

However, swapping makes 27 and 10 unsorted.

14 27 10 33 35 19 42 44

Hence, we swap them too.

14 10 27 33 35 19 42 44

Again we find 14 and 10 in an unsorted order.

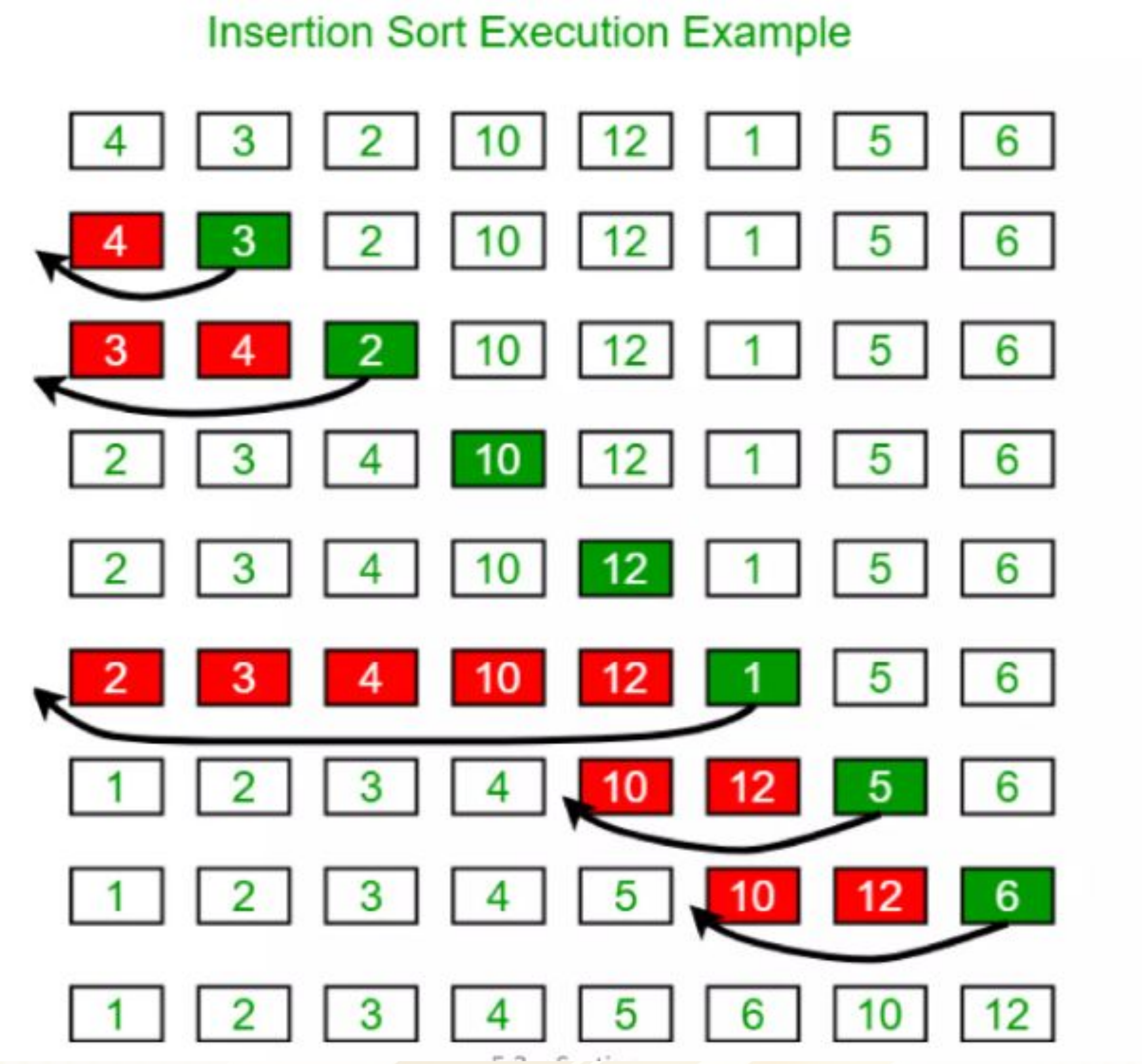
14 10 27 33 35 19 42 44

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

10 14 27 33 35 19 42 44

This process goes on until all the unsorted values are covered in a sorted sub-list.

Example:2



Insertion Sort Execution Example

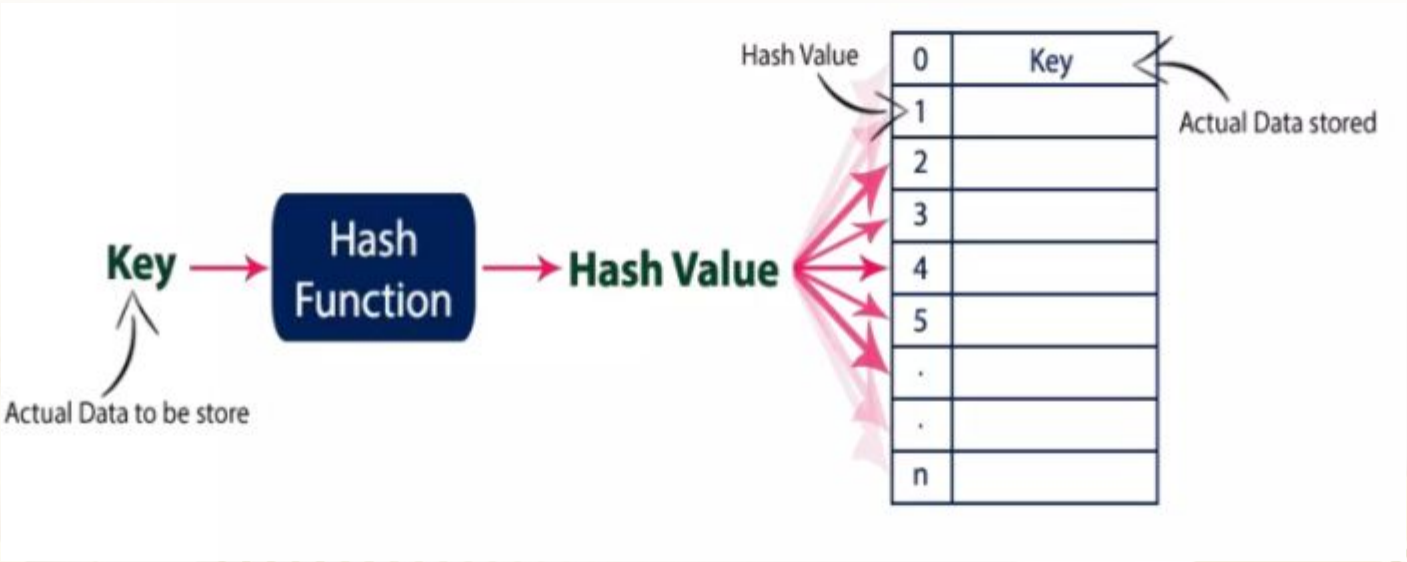
4	3	2	10	12	1	5	6
4	3	2	10	12	1	5	6
3	4	2	10	12	1	5	6
2	3	4	10	12	1	5	6
2	3	4	10	12	1	5	6
2	3	4	10	12	1	5	6
1	2	3	4	10	12	5	6
1	2	3	4	5	10	12	6
1	2	3	4	5	6	10	12

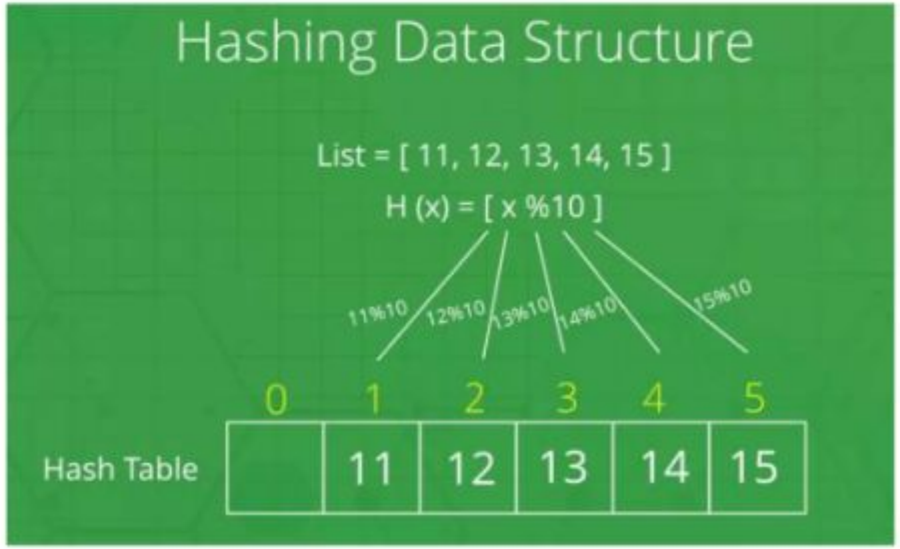
Hashing –Hash Tables

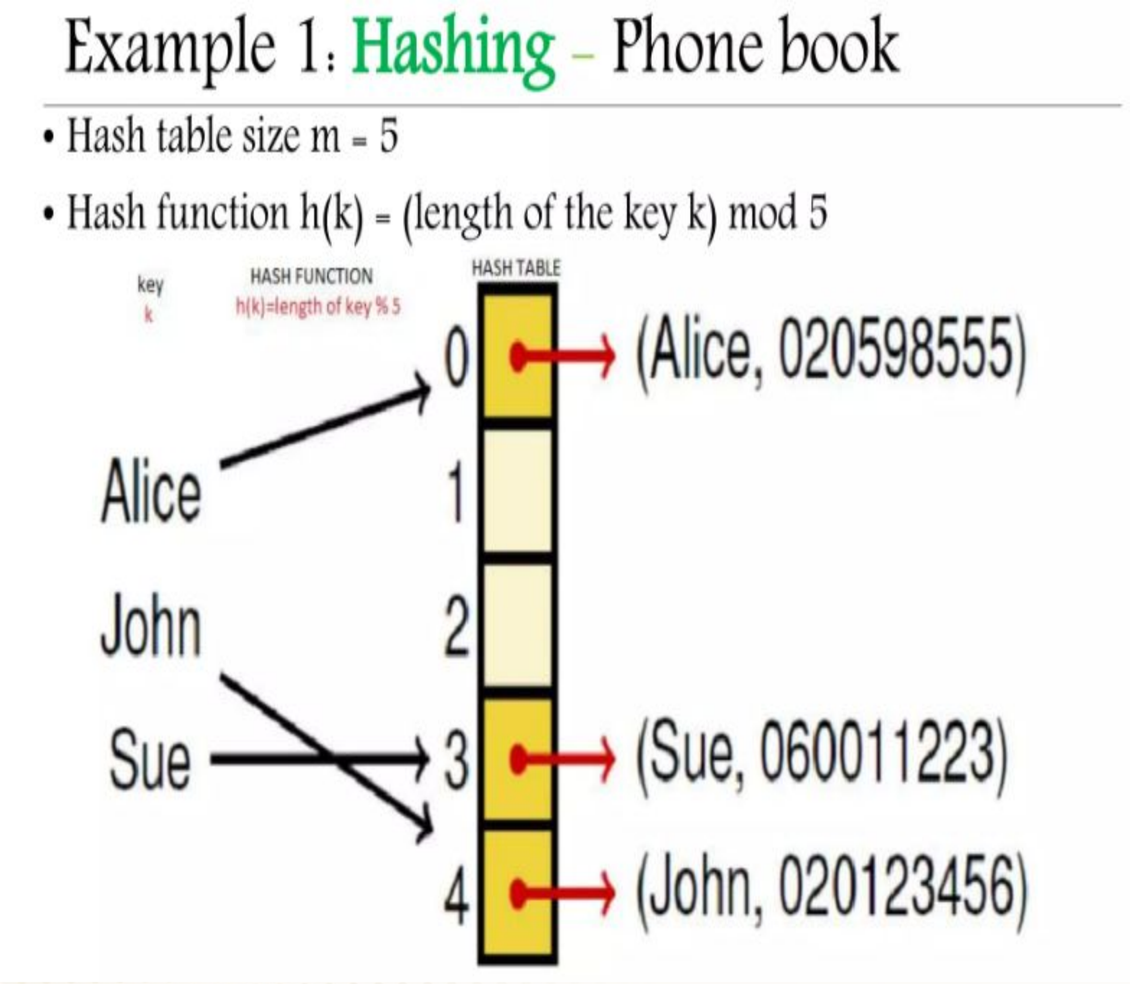
- Hashing in data structures is a technique used to map data (like keys) to a specific location (index) in a hash table using a hash function.
- The hash function computes an index, and the data is stored at that index, making data retrieval fast and efficient.
- In simple terms: Hashing turns a key into a unique index where the value is stored in a table.
- It allows for quick access to data, ideally in constant time, O(1).
- Implementation of hash table is frequently called hashing.
- Its technique used for performing insertions, deletions and finds in constant average time.
- Tree operations that require any ordering information among the elements that are not supported.
- Thus, operations such as find_min, find_max, and the printing of the entire table in sorted order in linear time are not cupported

Cont…

- Hashing is process of indexing and retrieving element(data) in a data structure to provide faster way of finding the element using a hash key or hash value generated using hash functions.
- Converting a given key into another value. Hash function is used generate the new value according to a mathematical algorithm. The result of hash function is known as hash value or simply hash.

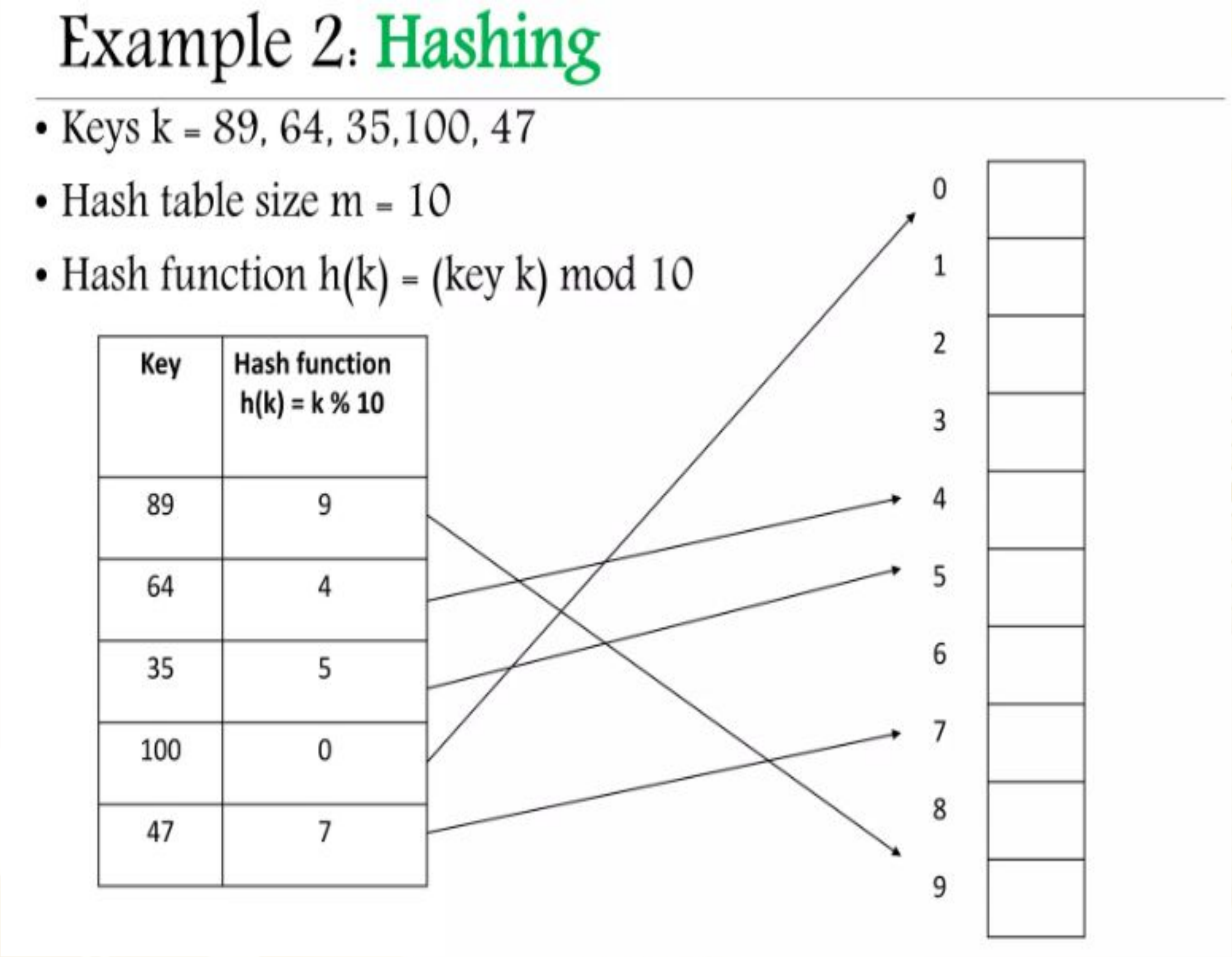






Example 1: Hashing – Phone book

- Hash table size m = 5
- Hash function h(k) = (length of the key k) mod 5



Example 2: Hashing

- Keys k = 89, 64, 35,100, 47
- Hash table size m = 10
- Hash function h(k) = (key k) mod 10

Key	Hash function h(k) = k % 10
89	9
64	4
35	5
100	0
47	7

Cont…

- Here, the central data structure is hash table. We will see
  - Several methods of implementing hash table
  - Compare these methods analytically
  - Show numerous applications of hashing
  - Compare hash table with binary search trees
- The ideal hash table data structure is merely an array of some fixed size, containing the keys.
- Typically, a key is string with an associated value(for example, salary information)
- We will refer to the table size as H_SIZE, with the understanding that this is a part of a hash data structure and not merely some variable floating around globally.
- The common convention is to have the table run from 0 to H_SIZE -1.
- Each key is mapped into some number in the range 0 to H_SIZE -1 and placed in the appropriate cell
- The mapping is called a hash function, which ideally should be simple to compute and should ensure that any two distinct keys get different cells.
- Since there are finite number of cells and virtually inexhaustible supply of keys, this is clearly impossible and thus we seek a hash function that distributes the keys even among the cells.

Hashing function

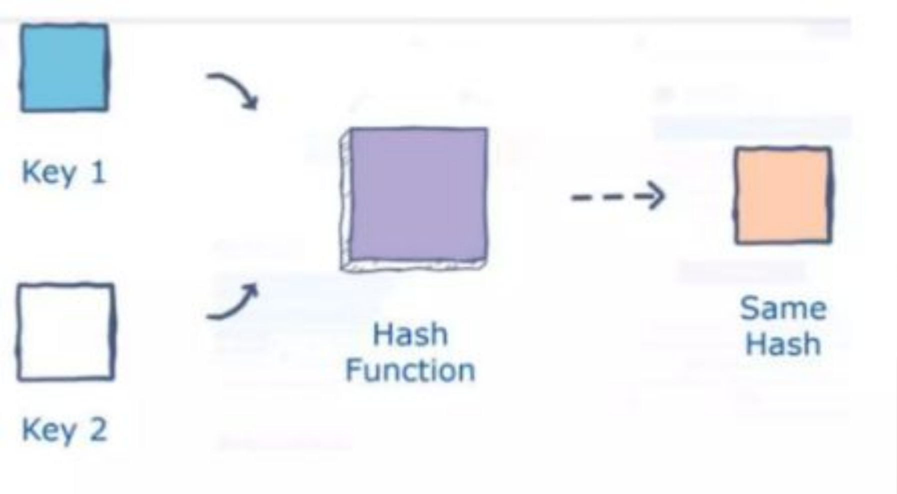
- If the input keys are integers, then simply returning key mod H_SIZE is generally a reasonable strategy, unless key happens to have some undesirable properties.
- In this case, the choice of hash function needs to be carefully considered.
- For example, if the table size is 10 and the keys all end in zero, then the standard hash function is obviously a bad choice.
- For a reasons we shall see later and to avoid situations like the one above, it is usually a good idea to ensure that the table size is prime.
- When the input keys are random integers, this function is not only very simple to compute but also distributes the keys evenly.
- Usually, the keys are strings in this case, the ash function needs to be chosen carefully.

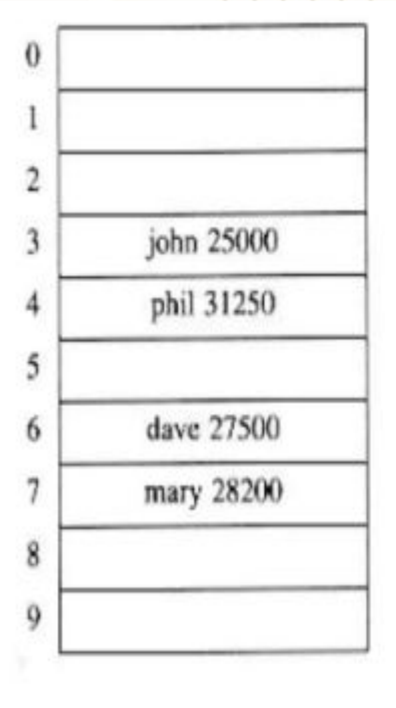
Example

- In this example, john hashes to 3, phil hashes to 4, dave hashes to 6 and mary hashes to 7

Collision

- The only remaining problems
  - Deal with choosing a function
  - Deciding what to do when two keys has to the same value(this is known as collision)
  - Deciding on the table size.

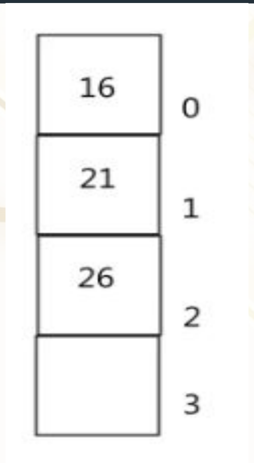




0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Cont...

- Consider the following example keys:(16,21,26,20)
- Size of the hash table is 3
- Now will start inserting the keys
    - For 16=16%4=0
    - 21=21%4=1
    - 26=26%4=2
    - 20=20%4=0
    - For the last key (20) there is already an element where 20 is to be inserted, this is where collision resolution comes into picture
- Collision resolution technique
    - Chaining(Open Hashing)
    - Open Addressing)(Closed Hashing)
        - Linear probing
        - Quadratic probing
        - Double hashing
- Keep in mind that two keys can generate the same hash. This phenomenon is known as collision
- Since hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in the same value.
- The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

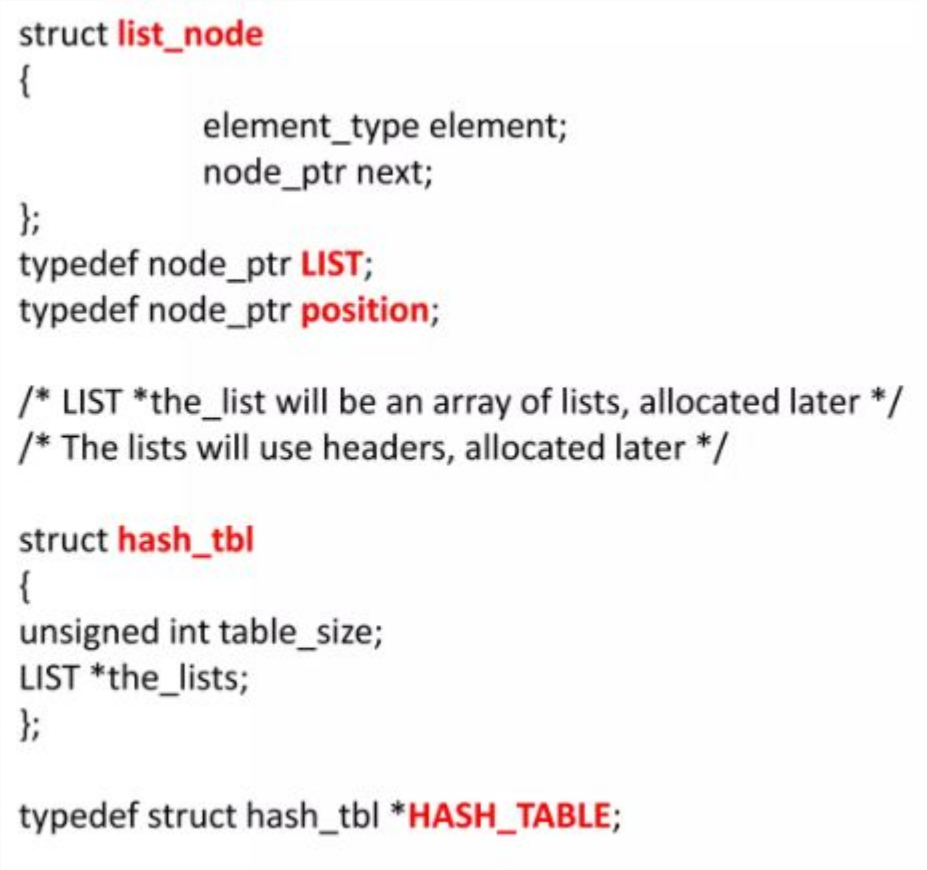


Open hashing(separate Chaining)

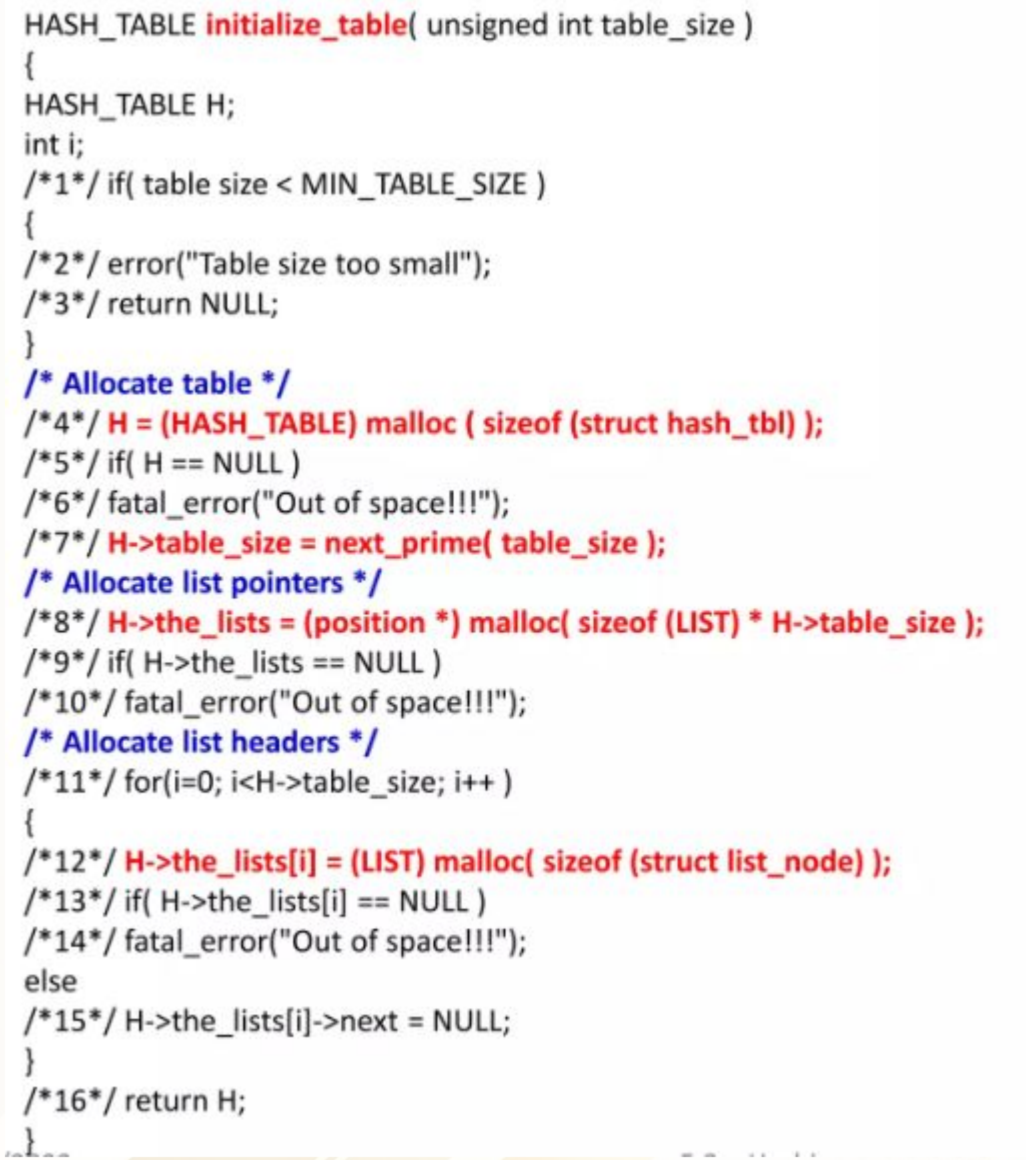
- Keep a list of all elements that hash to the same value.
- For convenience our lists have headers.
- If space is tight, it might be preferable to avoid their use
- **Open hashing Find and Insert**
- To perform find we use hash function to determine which list to traverse.
- Then traverse this list in the normal manner returning the position where the item is found
- To perform an insert we traverse down the appropriate list to check whether element is already in place
  - If duplicates are expected an extra field is usually kept and this field would be incremented in the event of match.
  - If the element turns out to be new, it is inserted either at the front of the list or at the end of list, whichever is easiest.

- This is an issue most easily addressed while the code is being written
- Sometimes new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be addressed in the near future

Open hashing declarations and Initialization

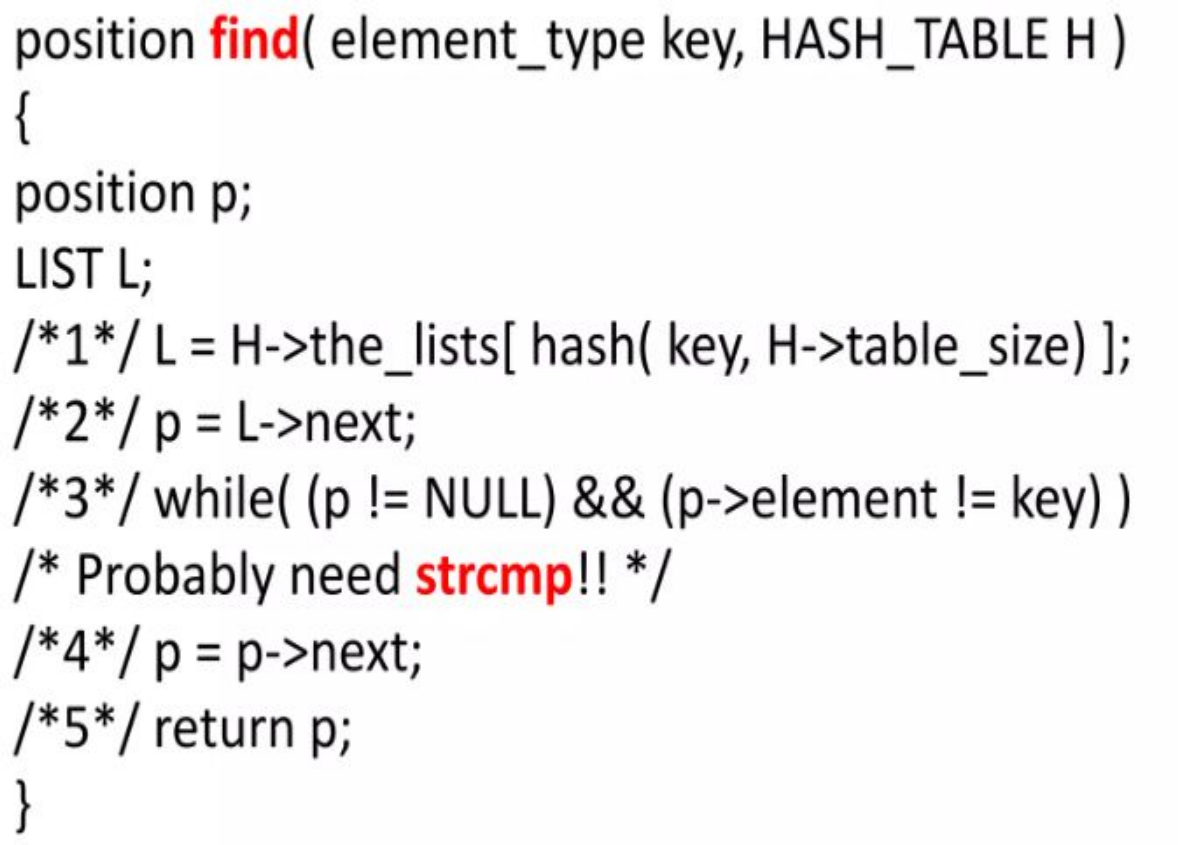


```
struct list_node
{
        element_type element;
        node_ptr next;
};
typedef node_ptr LIST;
typedef node_ptr position;

/* LIST *the_list will be an array of lists, allocated later */
/* The lists will use headers, allocated later */

struct hash_tbl
{
unsigned int table_size;
LIST *the_lists;
};

typedef struct hash_tbl *HASH_TABLE;
```

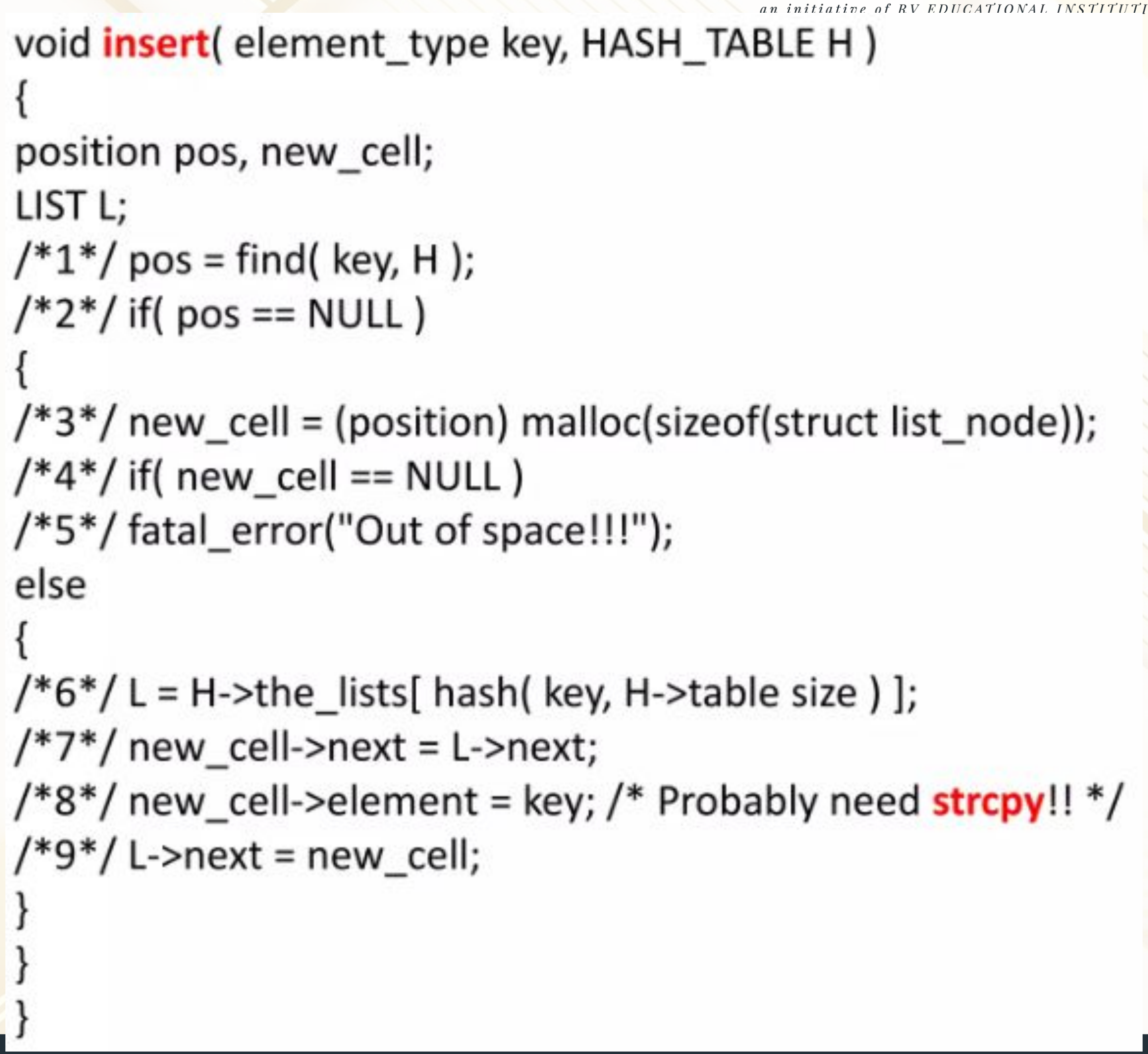


```
HASH_TABLE initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
/*1*/ if( table size < MIN_TABLE_SIZE )
{
/*2*/ error("Table size too small");
/*3*/ return NULL;
}
/* Allocate table */
/*4*/ H = (HASH_TABLE) malloc ( sizeof (struct hash_tbl) );
/*5*/ if( H == NULL )
/*6*/ fatal_error("Out of space!!!");
/*7*/ H->table_size = next_prime( table_size );
/* Allocate list pointers */
/*8*/ H->the_lists = (position *) malloc( sizeof (LIST) * H->table_size );
/*9*/ if( H->the_lists == NULL )
/*10*/ fatal_error("Out of space!!!");
/* Allocate list headers */
/*11*/ for(i=0; i<H->table_size; i++ )
{
/*12*/ H->the_lists[i] = (LIST) malloc( sizeof (struct list_node) );
/*13*/ if( H->the_lists[i] == NULL )
/*14*/ fatal_error("Out of space!!!");
else
/*15*/ H->the_lists[i]->next = NULL;
}
/*16*/ return H;
}
```

Open hashing find routine and insert routine



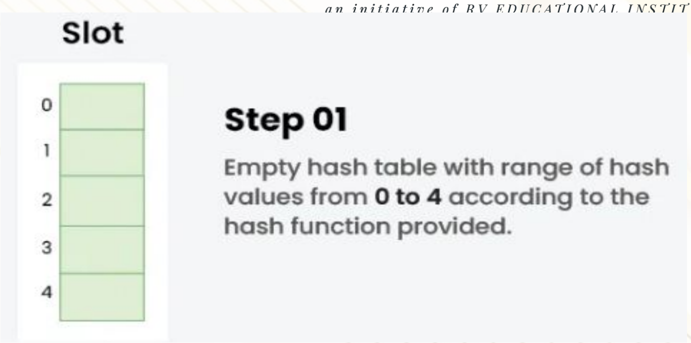
```
position find( element_type key, HASH_TABLE H )
{
position p;
LIST L;
/*1*/ L = H->the_lists[ hash( key, H->table_size) ];
/*2*/ p = L->next;
/*3*/ while( (p != NULL) && (p->element != key) )
/* Probably need strcmp!! */
/*4*/ p = p->next;
/*5*/ return p;
}
```



```
void insert( element_type key, HASH_TABLE H )
{
position pos, new_cell;
LIST L;
/*1*/ pos = find( key, H );
/*2*/ if( pos == NULL )
{
/*3*/ new_cell = (position) malloc(sizeof(struct list_node));
/*4*/ if( new_cell == NULL )
/*5*/ fatal_error("Out of space!!!");
else
{
/*6*/ L = H->the_lists[ hash( key, H->table size ) ];
/*7*/ new_cell->next = L->next;
/*8*/ new_cell->element = key; /* Probably need strcpy!! */
/*9*/ L->next = new_cell;
}
}
}
```

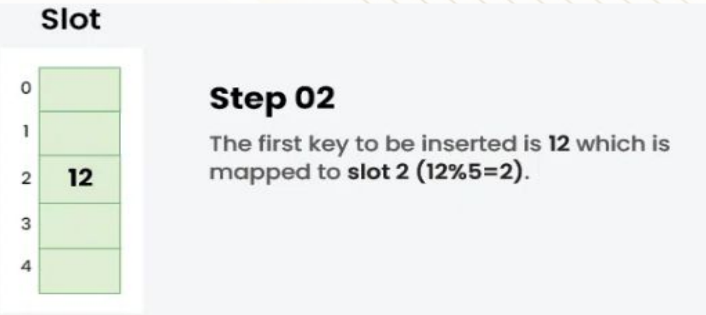
Open hashing Example

- Separate Chaining is a collision handling technique.
- Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.
- How to handle Collisions?
- There are mainly two methods to handle collision:
- Separate Chaining :
  - The idea behind separate chaining is to implement the array as a linked list called a chain.
  - *The **linked list** data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*
  - Let us consider a simple hash function as "**key mod 5**" and a sequence of keys as 12, 22, 15, 25



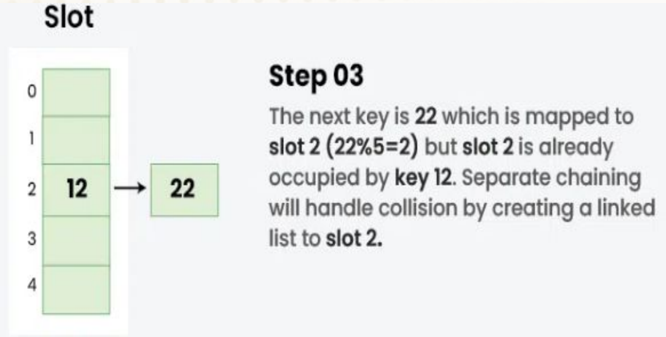
Step 01

Empty hash table with range of hash values from **0 to 4** according to the hash function provided.



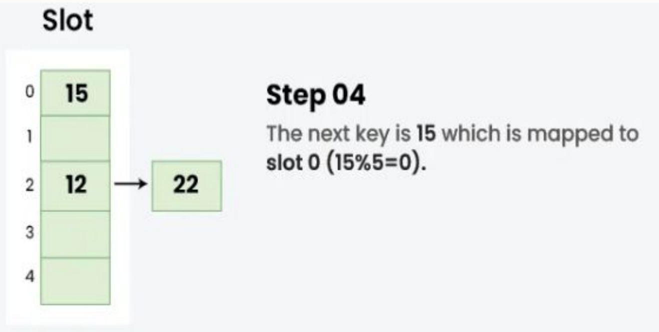
Step 02

The first key to be inserted is **12** which is mapped to **slot 2 (12%5=2)**.



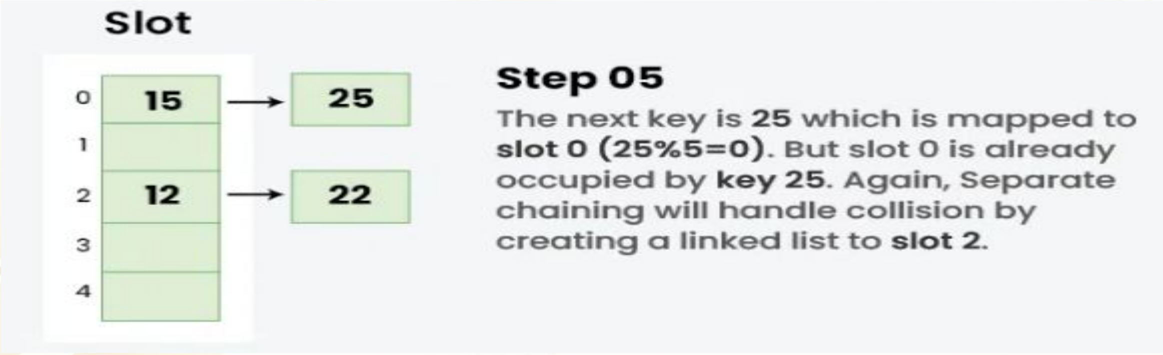
Step 03

The next key is **22** which is mapped to **slot 2 (22%5=2)** but **slot 2** is already occupied by **key 12**. Separate chaining will handle collision by creating a linked list to **slot 2.**



Step 04

The next key is **15** which is mapped to **slot 0 (15%5=0)**.



Step 05

The next key is **25** which is mapped to **slot 0 (25%5=0)**. But slot 0 is already occupied by **key 25**. Again, Separate chaining will handle collision by creating a linked list to **slot 2**.

Closed Hashing(open addressing)

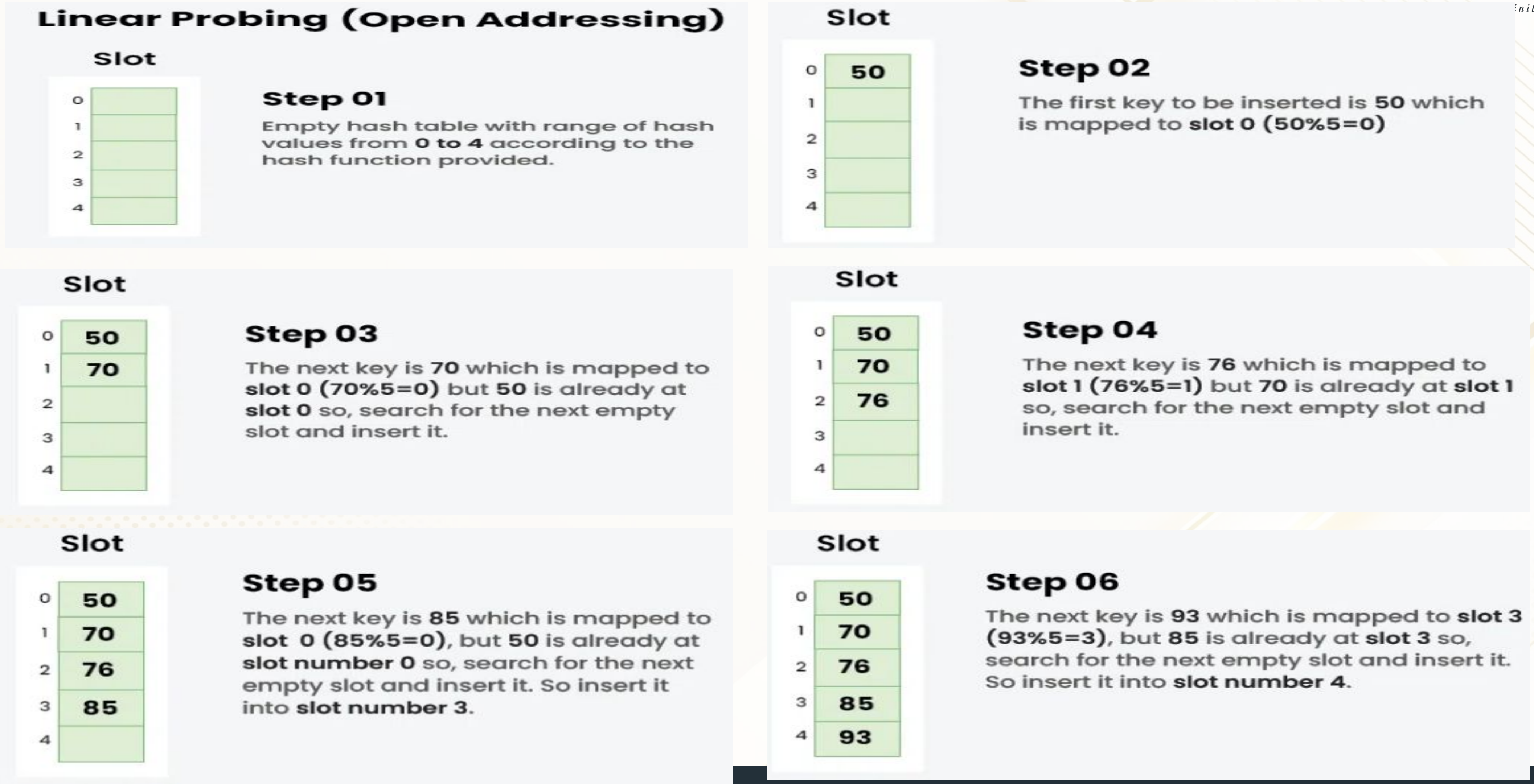
- Open addressing has the disadvantage of requiring pointer
- Open Addressing is a method for handling collisions.
- In Open Addressing, all elements are stored in the hash table itself.
- So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing.
- This entire procedure is based upon probing.

Collision handling strategies

- 3 common collision strategies
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- Linear Probing
- In linear probing, the hash table is searched sequentially that starts from the original location of the hash.
- If in case the location that we get is already occupied, then we check for the next location.
- The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.
- Example:The typical gap between two probes is 1 as seen in the example below:
- Let hash(x) be the slot index computed using a hash function and S be the table size
- If slot hash(x) % S is full, then we try (hash(x) + 1) % S
- If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
- If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

Linear probing example

- Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.



Linear Probing (Open Addressing)

Slot	
0	
1	
2	
3	
4	

Step 01

Empty hash table with range of hash values from **0 to 4** according to the hash function provided.

Slot	
0	50
1	
2	
3	
4	

Step 02

The first key to be inserted is **50** which is mapped to **slot 0 (50%5=0)**

Slot	
0	50
1	70
2	
3	
4	

Step 03

The next key is **70** which is mapped to **slot 0 (70%5=0)** but **50** is already at **slot 0** so, search for the next empty slot and insert it.

Slot	
0	50
1	70
2	76
3	
4	

Step 04

The next key is **76** which is mapped to **slot 1 (76%5=1)** but **70** is already at **slot 1** so, search for the next empty slot and insert it.

Slot	
0	50
1	70
2	76
3	85
4	

Step 05

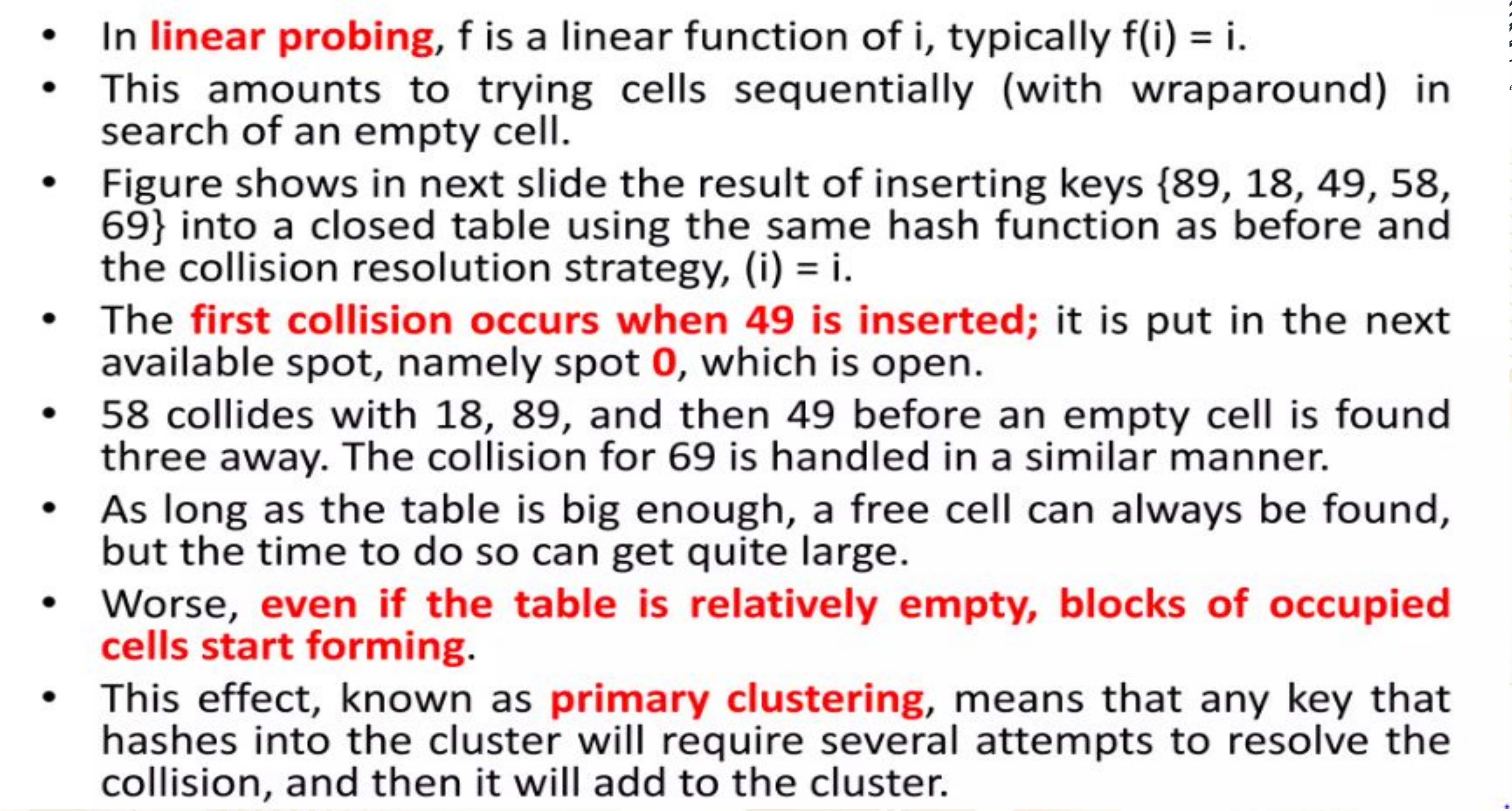
The next key is **85** which is mapped to **slot 0 (85%5=0)**, but **50** is already at **slot number 0** so, search for the next empty slot and insert it. So insert it into **slot number 3**.

Slot	
0	50
1	70
2	76
3	85
4	93

Step 06

The next key is **93** which is mapped to **slot 3 (93%5=3)**, but **85** is already at **slot 3** so, search for the next empty slot and insert it. So insert it into **slot number 4**.

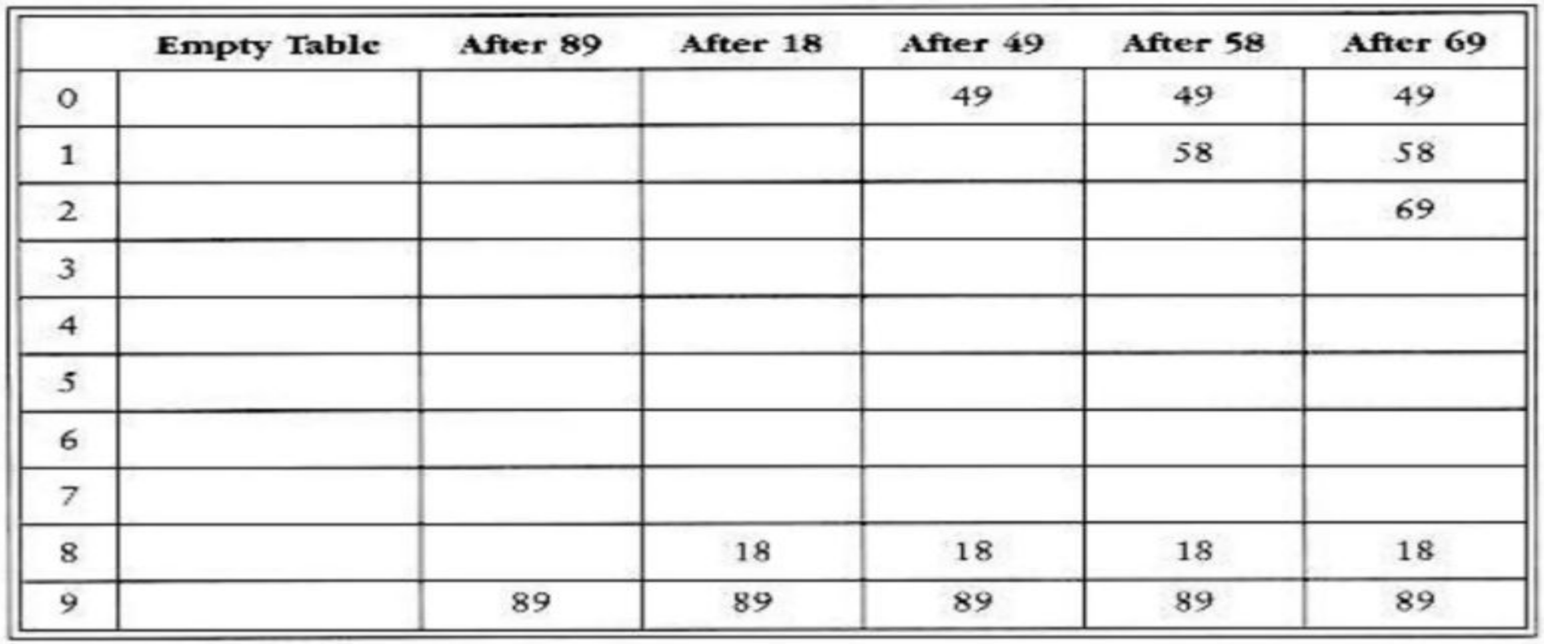
Cont..



- In **linear probing**, f is a linear function of i, typically f(i) = i.
- This amounts to trying cells sequentially (with wraparound) in search of an empty cell.
- Figure shows in next slide the result of inserting keys {89, 18, 49, 58, 69} into a closed table using the same hash function as before and the collision resolution strategy, (i) = i.
- The **first collision occurs when 49 is inserted;** it is put in the next available spot, namely spot **0**, which is open.
- 58 collides with 18, 89, and then 49 before an empty cell is found three away. The collision for 69 is handled in a similar manner.
- As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, **even if the table is relatively empty, blocks of occupied cells start forming**.
- This effect, known as **primary clustering**, means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Example

- Although we will not perform the calculations here, it can be shown that the expected **number of probes** using **linear probing** is roughly
  - **$1/2(1 + 1/(1 - \Delta)^2)$** for **insertions** and **unsuccessful searches** and
  - **$1/2(1 + 1/ (1- \Delta))$** for **successful searches**.

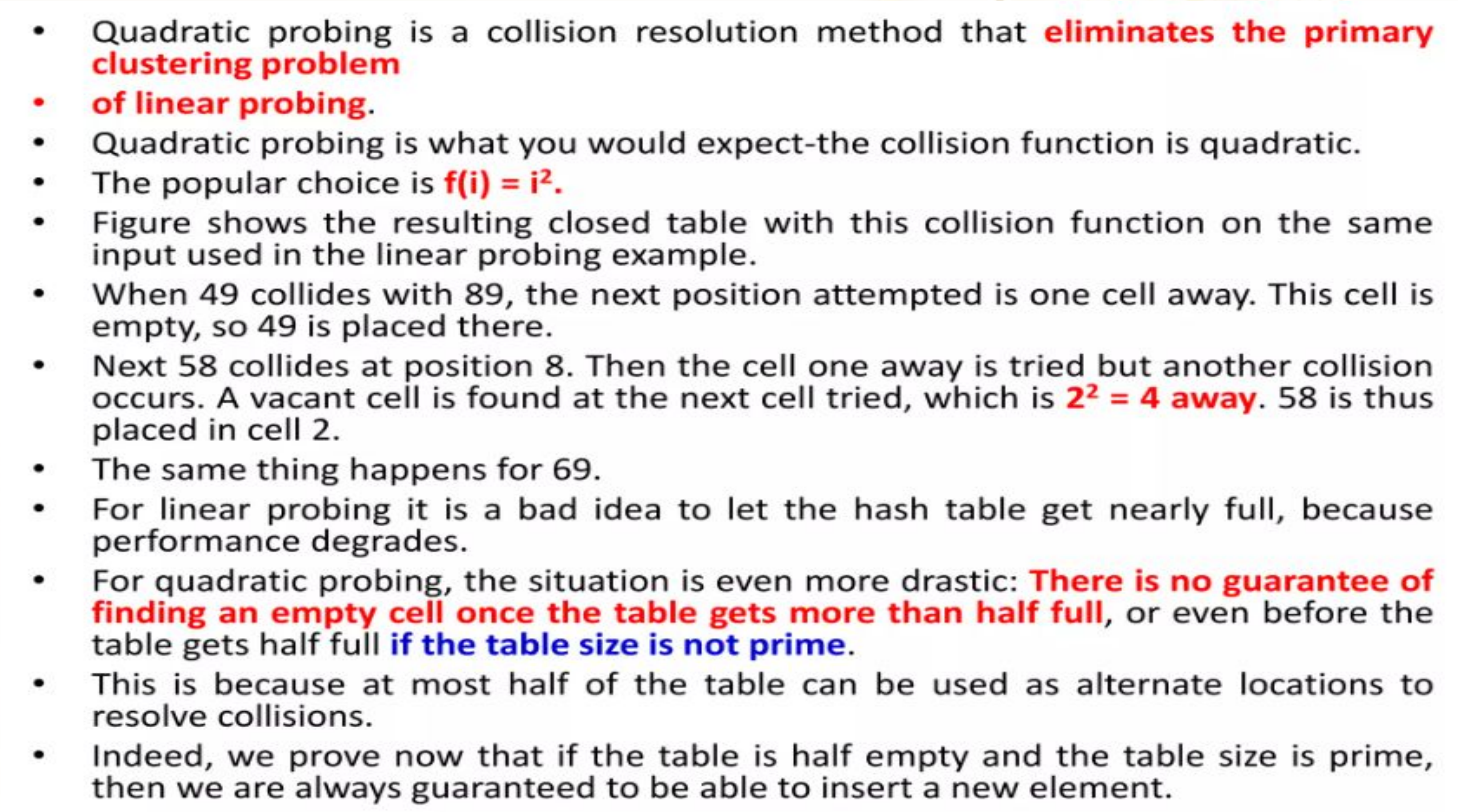


	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic Probing

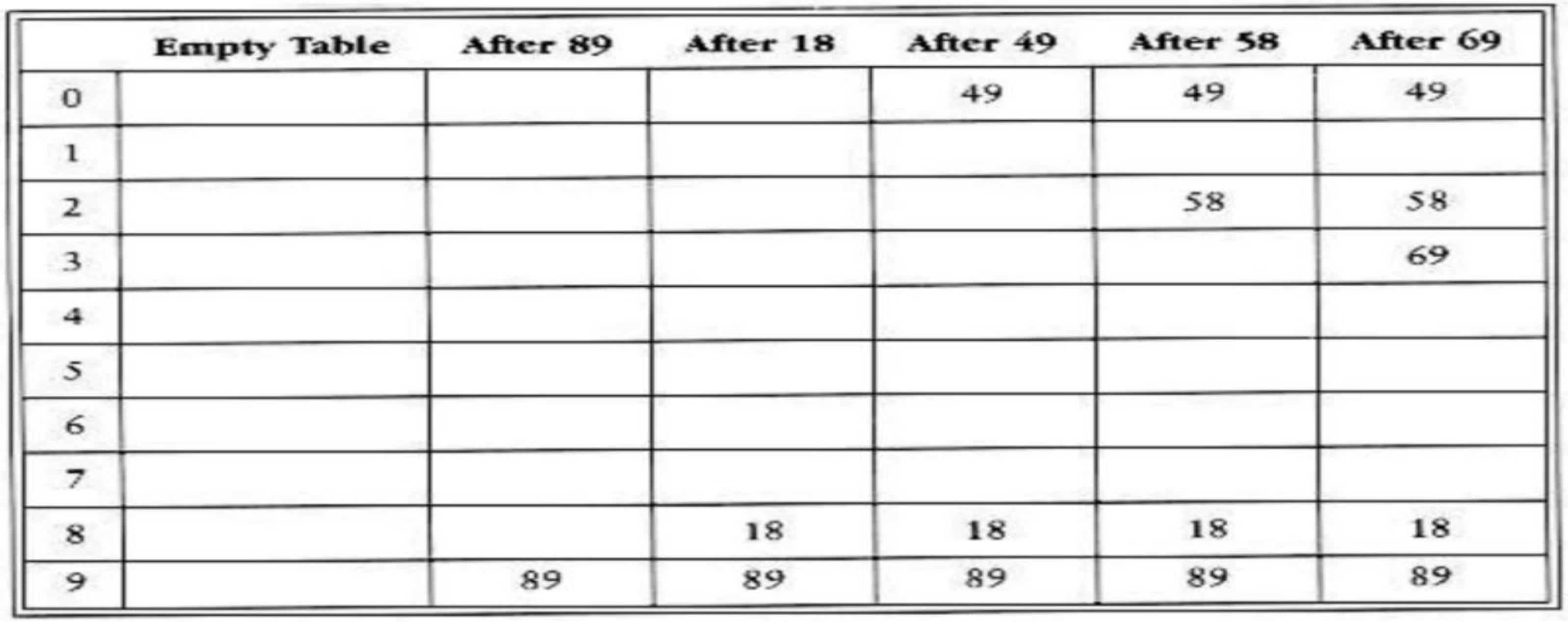
- If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value.
- Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above.
- This method is also known as the mid-square method.
- In this method, we look for the i2‘th slot in the ith iteration.
- We always start from the original hash location. If only the location is occupied then we check the other slots.
- let hash(x) be the slot index computed using hash function.
- If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
- If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
- If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Quadratic Probing



- Quadratic probing is a collision resolution method that **eliminates the primary clustering problem**
- **of linear probing**.
- Quadratic probing is what you would expect-the collision function is quadratic.
- The popular choice is **$f(i) = i^2$.**
- Figure shows the resulting closed table with this collision function on the same input used in the linear probing example.
- When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there.
- Next 58 collides at position 8. Then the cell one away is tried but another collision occurs. A vacant cell is found at the next cell tried, which is **$2^2 = 4$ away**. 58 is thus placed in cell 2.
- The same thing happens for 69.
- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.
- For quadratic probing, the situation is even more drastic: **There is no guarantee of finding an empty cell once the table gets more than half full**, or even before the table gets half full **if the table size is not prime**.
- This is because at most half of the table can be used as alternate locations to resolve collisions.
- Indeed, we prove now that if the table is half empty and the table size is prime, then we are always guaranteed to be able to insert a new element.

Example

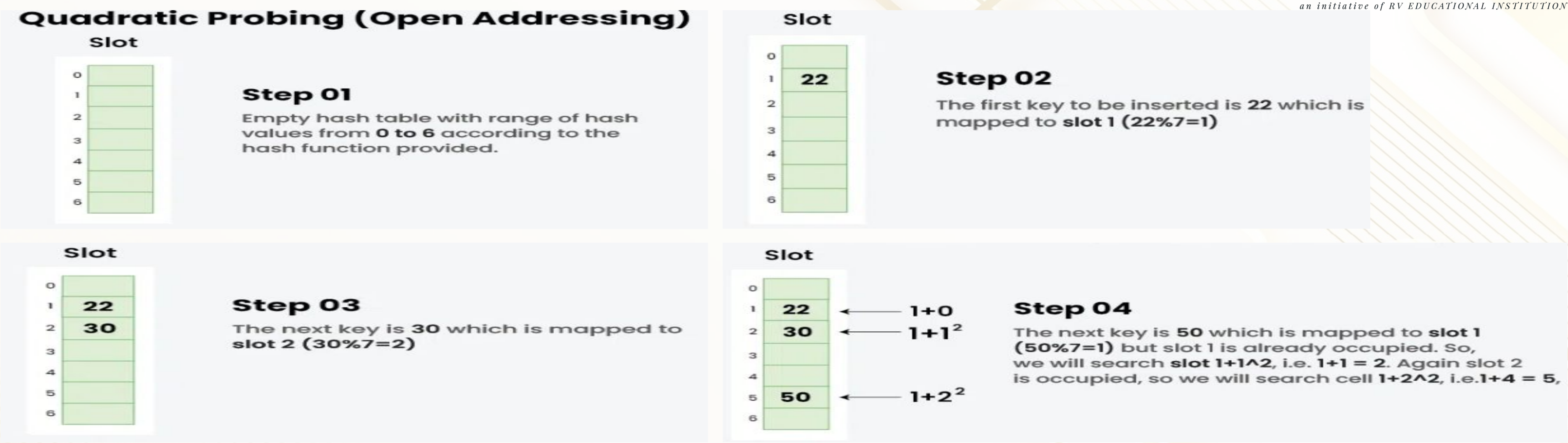


	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

- Although quadratic probing **eliminates primary clustering,** elements that hash to the **same position will probe the same alternate cells**. This is known as **secondary clustering**.
- Secondary clustering is a **slight theoretical blemish**.
- Simulation results suggest that it generally causes less than an extra probe per search.
- **Double hashing technique eliminates this**, but does so at the cost of **extra multiplications** and **divisions**.

Cont..

- Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.



Quadratic Probing (Open Addressing)

Slot	
0	
1	
2	
3	
4	
5	
6	

Step 01

Empty hash table with range of hash values from **0 to 6** according to the hash function provided.

Slot	
0	
1	22
2	
3	
4	
5	
6	

Step 02

The first key to be inserted is **22** which is mapped to **slot 1 (22%7=1)**

Slot	
0	
1	22
2	30
3	
4	
5	
6	

Step 03

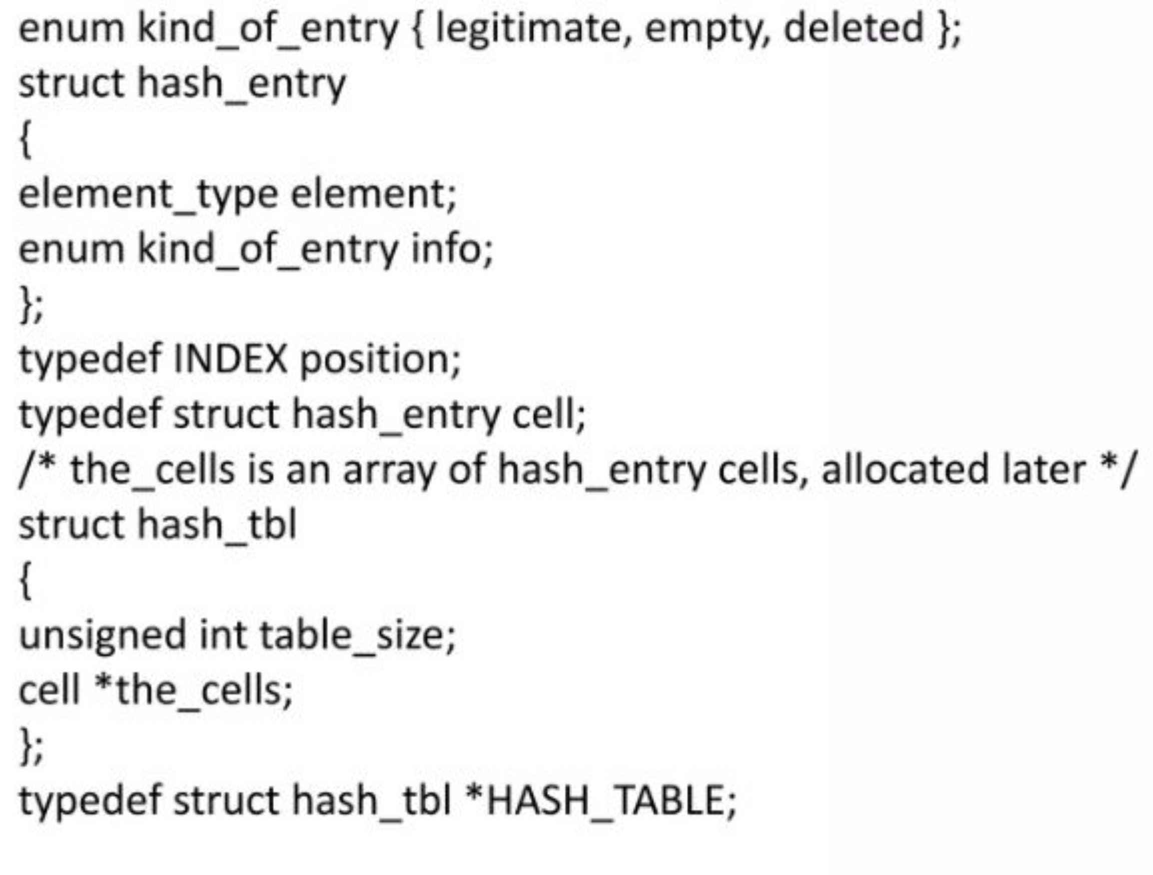
The next key is **30** which is mapped to **slot 2 (30%7=2)**

Slot		
0		
1	22	← 1+0
2	30	← $1+1^2$
3		
4		
5	50	← $1+2^2$
6		

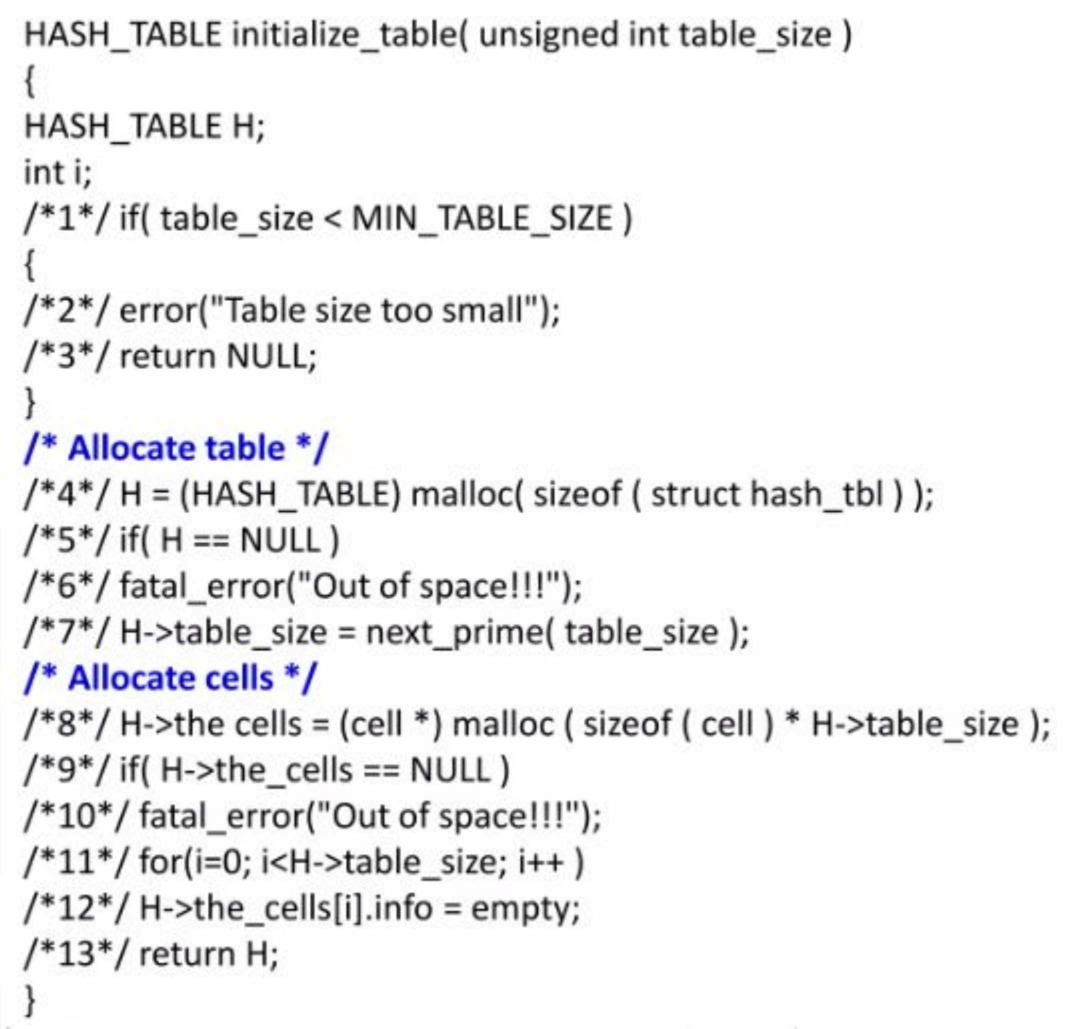
Step 04

The next key is **50** which is mapped to **slot 1 (50%7=1)** but slot 1 is already occupied. So, we will search **slot 1+1^2**, i.e. **1+1 = 2**. Again slot 2 is occupied, so we will search cell **1+2^2**, i.e.**1+4 = 5**,

Closed hashing –Type declaration and initialization

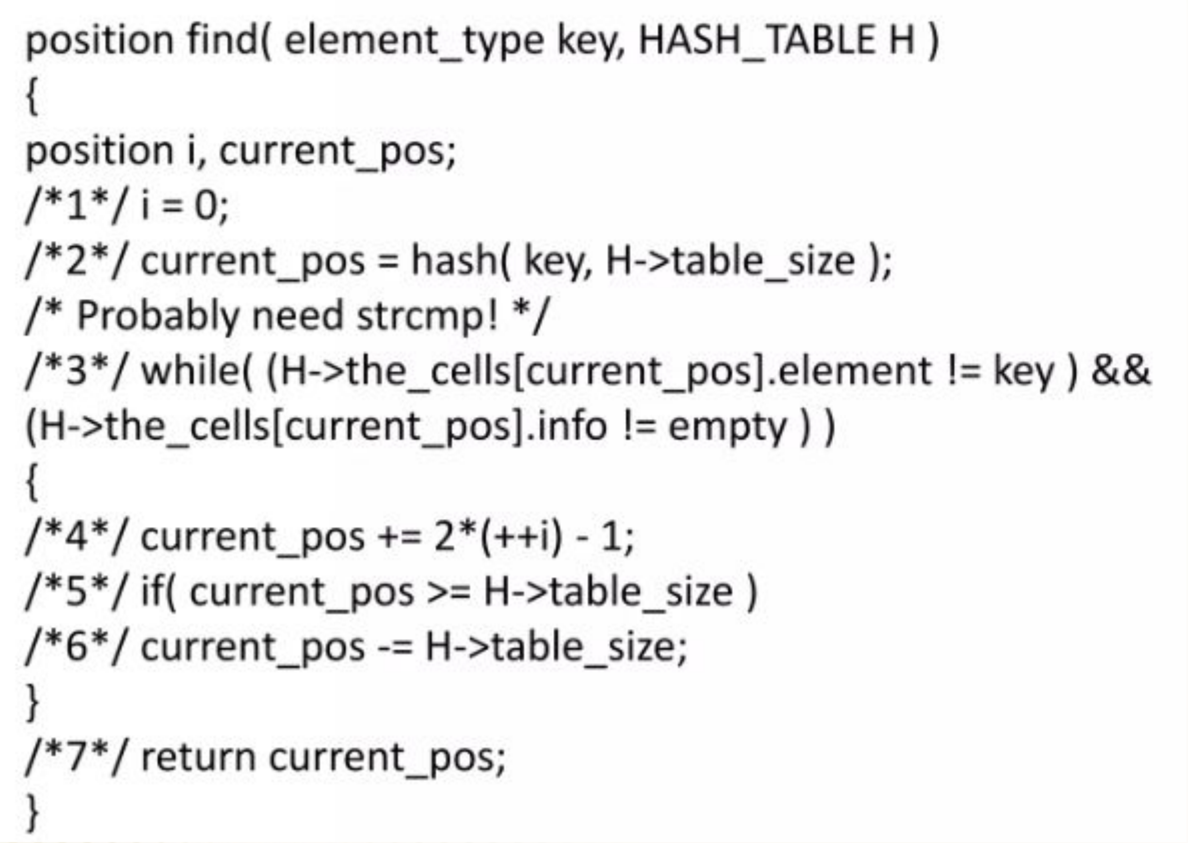


```
enum kind_of_entry { legitimate, empty, deleted };
struct hash_entry
{
element_type element;
enum kind_of_entry info;
};
typedef INDEX position;
typedef struct hash_entry cell;
/* the_cells is an array of hash_entry cells, allocated later */
struct hash_tbl
{
unsigned int table_size;
cell *the_cells;
};
typedef struct hash_tbl *HASH_TABLE;
```



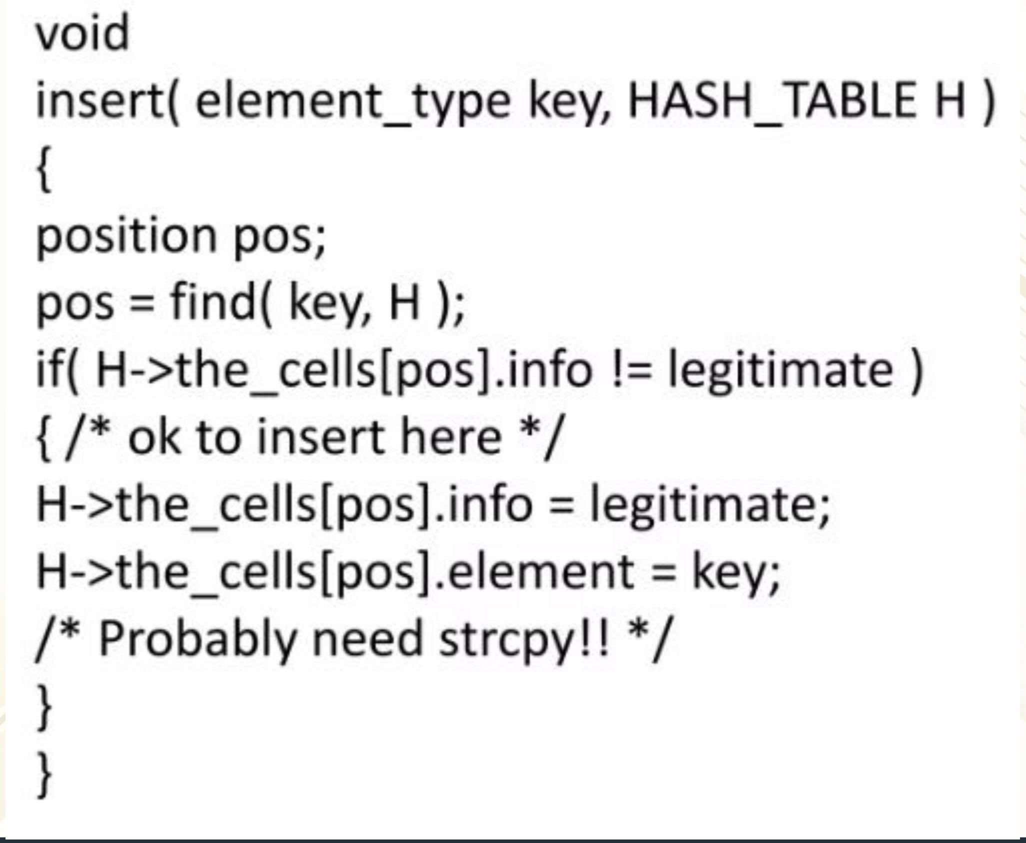
```
HASH_TABLE initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
/*1*/ if( table_size < MIN_TABLE_SIZE )
{
/*2*/ error("Table size too small");
/*3*/ return NULL;
}
/* Allocate table */
/*4*/ H = (HASH_TABLE) malloc( sizeof ( struct hash_tbl ) );
/*5*/ if( H == NULL )
/*6*/ fatal_error("Out of space!!!");
/*7*/ H->table_size = next_prime( table_size );
/* Allocate cells */
/*8*/ H->the cells = (cell *) malloc ( sizeof ( cell ) * H->table_size );
/*9*/ if( H->the_cells == NULL )
/*10*/ fatal_error("Out of space!!!");
/*11*/ for(i=0; i<H->table_size; i++ )
/*12*/ H->the_cells[i].info = empty;
/*13*/ return H;
}
```

Closed hashing –Find routine with Quadratic probing



```
position find( element_type key, HASH_TABLE H )
{
position i, current_pos;
/*1*/ i = 0;
/*2*/ current_pos = hash( key, H->table_size );
/* Probably need strcmp! */
/*3*/ while( (H->the_cells[current_pos].element != key ) &&
(H->the_cells[current_pos].info != empty ) )
{
/*4*/ current_pos += 2*(++i) - 1;
/*5*/ if( current_pos >= H->table_size )
/*6*/ current_pos -= H->table_size;
}
/*7*/ return current_pos;
}
```

Closed hashing –insert routine with Quadratic probing



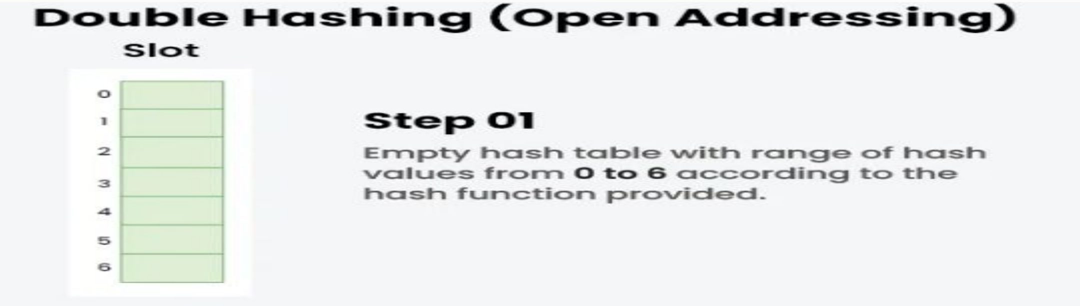
```
void
insert( element_type key, HASH_TABLE H )
{
position pos;
pos = find( key, H );
if( H->the_cells[pos].info != legitimate )
{ /* ok to insert here */
H->the_cells[pos].info = legitimate;
H->the_cells[pos].element = key;
/* Probably need strcpy!! */
}
}
```

Double hashing

- The intervals that lie between probes are computed by another hash function.
- Double hashing is a technique that reduces clustering in an optimized way.
- In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the ith rotation.
- let hash(x) be the slot index computed using hash function.
- If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
- If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
- If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Example

- Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is h1(k) = k mod 7 and second hash-function is h2(k) = 1 + (k mod 5)

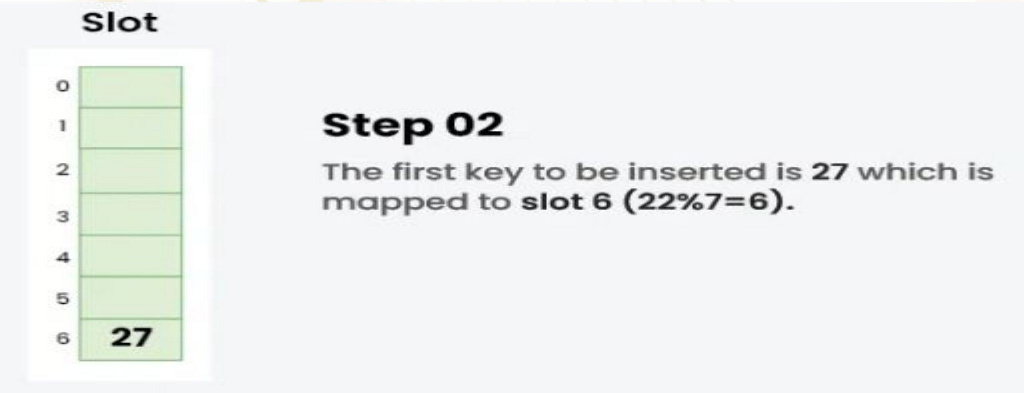


Double Hashing (Open Addressing)

Slot	
0	
1	
2	
3	
4	
5	
6	

Step 01

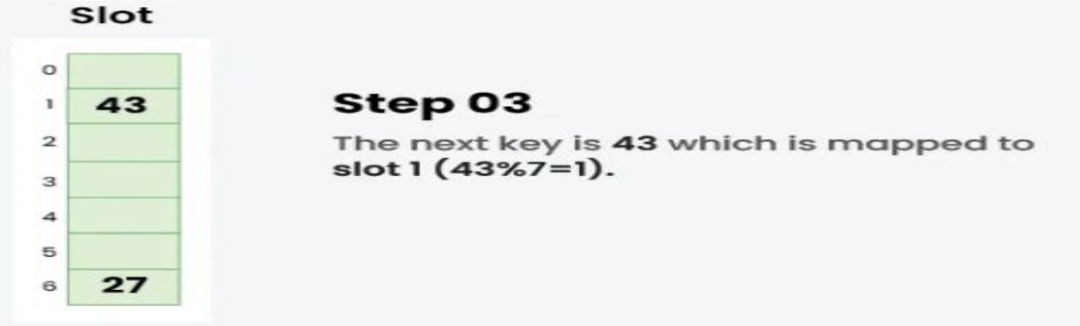
Empty hash table with range of hash values from **0 to 6** according to the hash function provided.



Slot	
0	
1	
2	
3	
4	
5	
6	27

Step 02

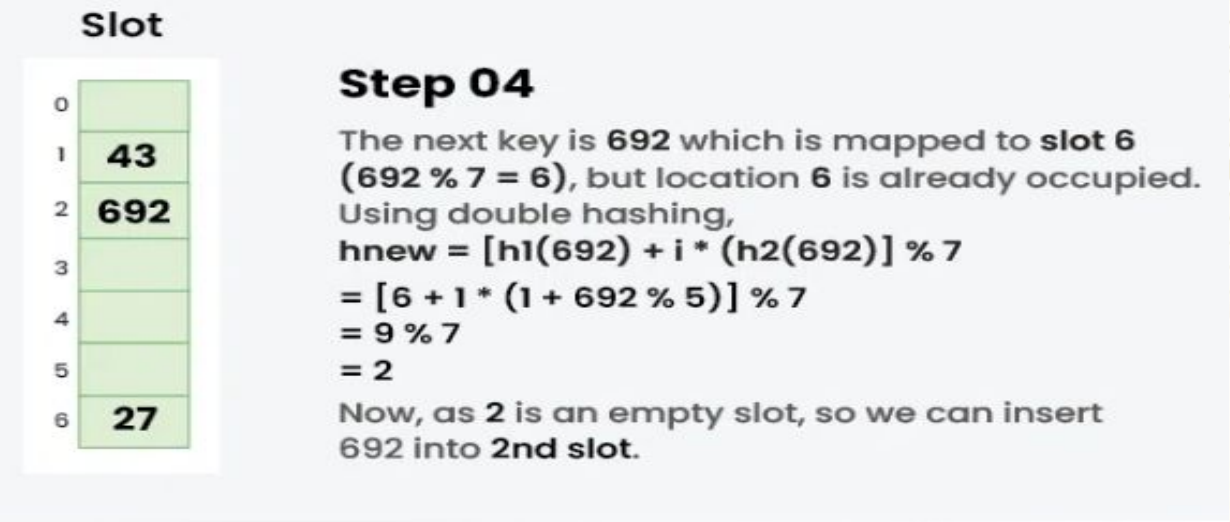
The first key to be inserted is **27** which is mapped to **slot 6 (22%7=6).**



Slot	
0	
1	43
2	
3	
4	
5	
6	27

Step 03

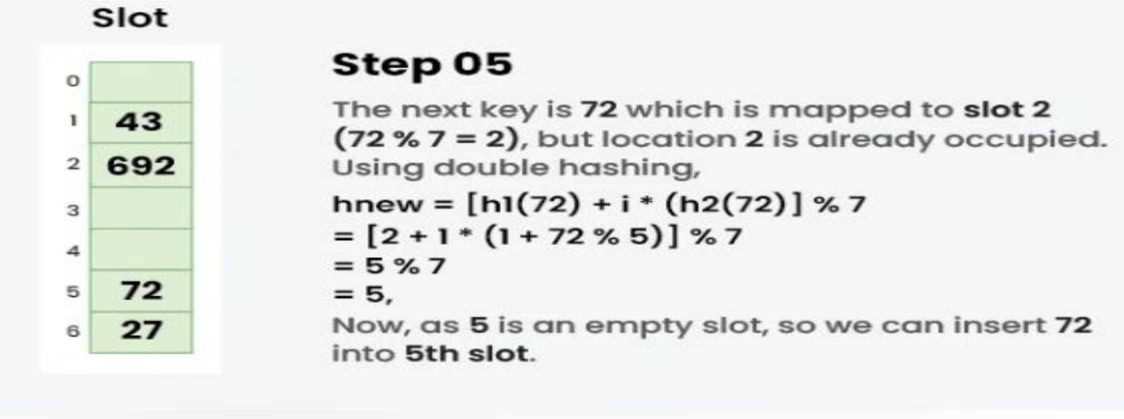
The next key is **43** which is mapped to **slot 1 (43%7=1).**



Slot	
0	
1	43
2	692
3	
4	
5	
6	27

Step 04

The next key is **692** which is mapped to **slot 6 (692 % 7 = 6)**, but location **6** is already occupied.
Using double hashing,
hnew = [h1(692) + i * (h2(692)] % 7
= [6 + 1 * (1 + 692 % 5)] % 7
= 9 % 7
= 2
Now, as **2** is an empty slot, so we can insert 692 into **2nd slot.**

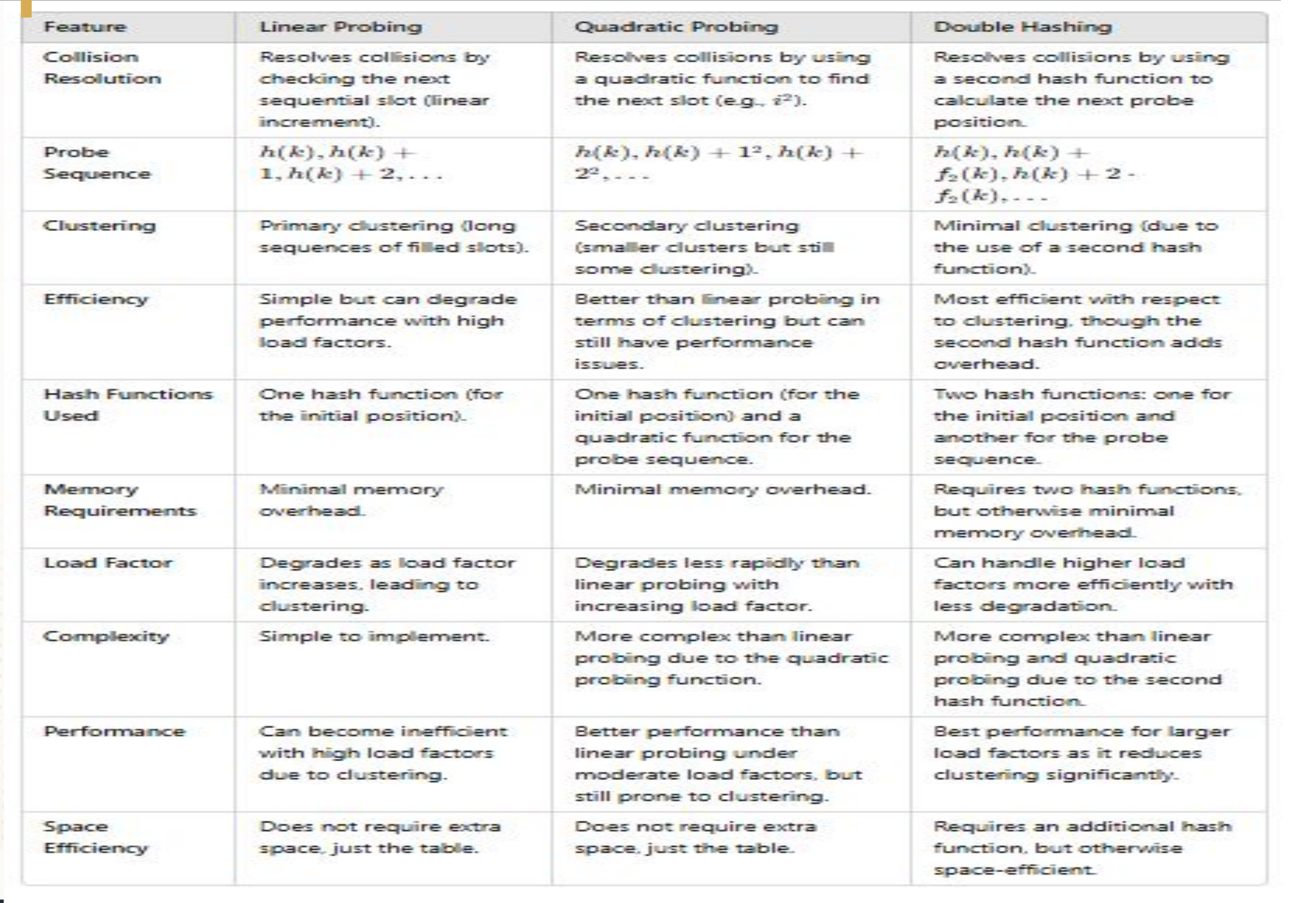


Slot	
0	
1	43
2	692
3	
4	
5	72
6	27

Step 05

The next key is **72** which is mapped to **slot 2 (72 % 7 = 2)**, but location **2** is already occupied.
Using double hashing,
hnew = [h1(72) + i * (h2(72)] % 7
= [2 + 1 * (1 + 72 % 5)] % 7
= 5 % 7
= 5,
Now, as **5** is an empty slot, so we can insert **72** into **5th slot.**

Comparison



Feature	Linear Probing	Quadratic Probing	Double Hashing
Collision Resolution	Resolves collisions by checking the next sequential slot (linear increment).	Resolves collisions by using a quadratic function to find the next slot (e.g., i^2).	Resolves collisions by using a second hash function to calculate the next probe position.
Probe Sequence	$h(k), h(k) + 1, h(k) + 2, \ldots$	$h(k), h(k) + 1^2, h(k) + 2^2, \ldots$	$h(k), h(k) + f_2(k), h(k) + 2 \cdot f_2(k), \ldots$
Clustering	Primary clustering (long sequences of filled slots).	Secondary clustering (smaller clusters but still some clustering).	Minimal clustering (due to the use of a second hash function).
Efficiency	Simple but can degrade performance with high load factors.	Better than linear probing in terms of clustering but can still have performance issues.	Most efficient with respect to clustering, though the second hash function adds overhead.
Hash Functions Used	One hash function (for the initial position).	One hash function (for the initial position) and a quadratic function for the probe sequence.	Two hash functions: one for the initial position and another for the probe sequence.
Memory Requirements	Minimal memory overhead.	Minimal memory overhead.	Requires two hash functions, but otherwise minimal memory overhead.
Load Factor	Degrades as load factor increases, leading to clustering.	Degrades less rapidly than linear probing with increasing load factor.	Can handle higher load factors more efficiently with less degradation.
Complexity	Simple to implement.	More complex than linear probing due to the quadratic probing function.	More complex than linear probing and quadratic probing due to the second hash function.
Performance	Can become inefficient with high load factors due to clustering.	Better performance than linear probing under moderate load factors, but still prone to clustering.	Best performance for larger load factors as it reduces clustering significantly.
Space Efficiency	Does not require extra space, just the table.	Does not require extra space, just the table.	Requires an additional hash function, but otherwise space-efficient.

Reference links

- https://www.slideshare.net/jaravles/searching-sorting-and-hashing-techniques

Thank You