Sri Eshwar College of Engineering
An Autonomous Institution
Affiliated to Anna University, Chennai

Celebrating 14 Years of Excellence

Data Structures (Module 5) – Sorting

Sorting

- Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

- **Types of sorting:**
- Bubble Sort
- Insertion Sort
- Quick Sort
- Merge Sort

BUBBLE SORT

Bubble Sort

- Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. The array elements in bubble sort move to the end in each iteration.

Algorithm:

Step 1: Compare the first two items in the list. If they are in the correct order, leave them. If items are not in the correct order, swap them.

Step 2: Repeat the step 1 process for items '2' and '3' in the list.

Step 3: Continue applying the step 1 process to the rest of the items in the list.

Step 4: Once one pass is completed, repeat the process again and again until no swaps are required and all the numbers are in the correct order.

Bubble Sort

Time Complexity of the Bubble Sort Algorithm:
Although it is simple to use, the performance of bubble sort is poor. It is not suitable for large data sets.

The average and worst-case complexity of Bubble sort is **$O(n^2)$**, where 'n' is a number of items.

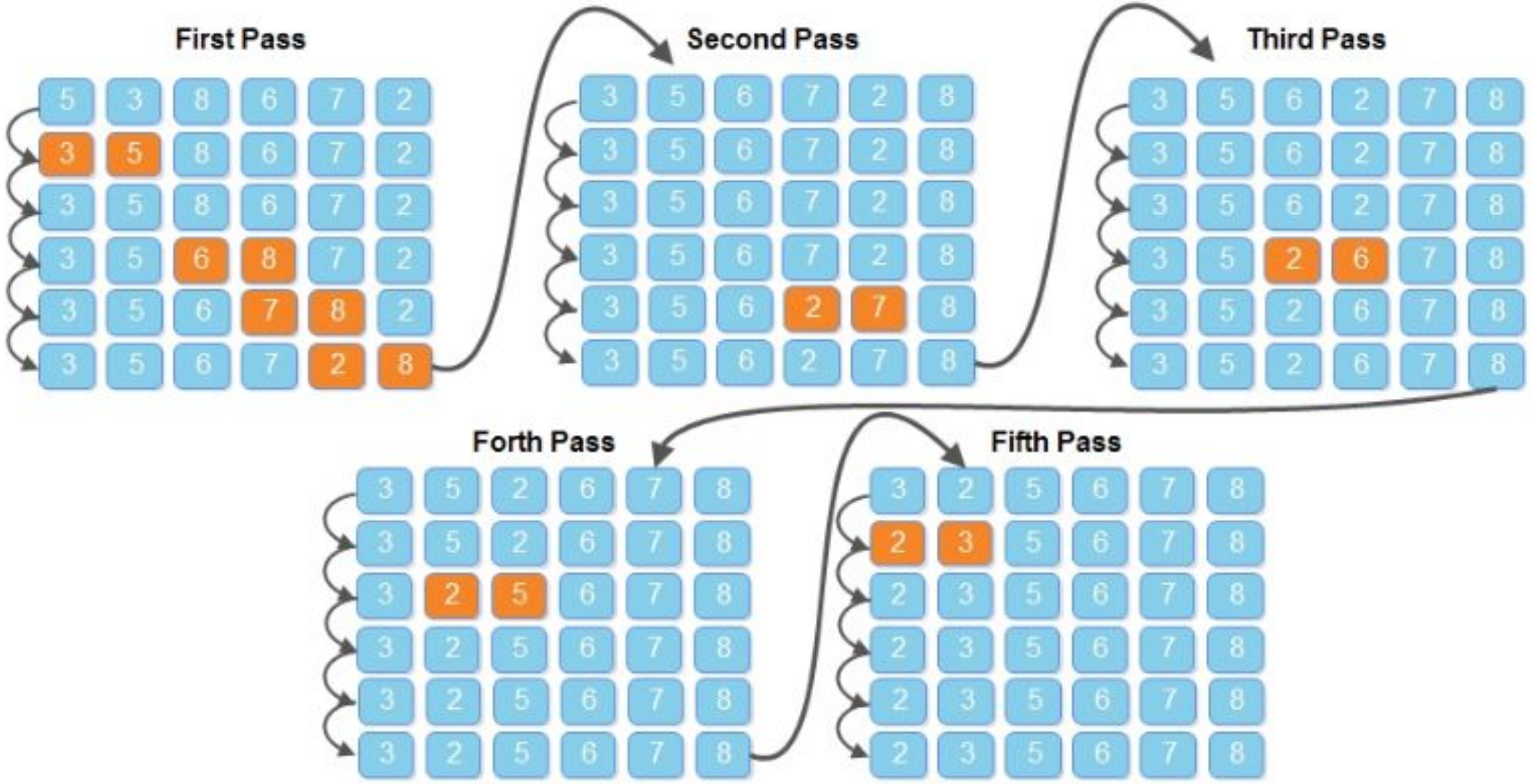
Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(n^2)$

Bubble Sort:

Example:



Bubble Sort: Program

```
#include<stdio.h>
#define size 4
int A[size];
void BUBBLE_SORT(int A[]);
void main()
{
   int i;
   printf("Enter the values to be sorted:\n");
   for(i = 0; i <= size-1; i++)
   {
      scanf("%d", &A[i]);
   }
   BUBBLE_SORT(A);
   printf("After bubble sort:\n");

   for(i = 0; i <= size-1; i++)
   {
      printf("%d ", A[i]);
   }
}
```

```
void BUBBLE_SORT(int A[])
{
   int i, j, temp;
   for(i = 0; i < size-1; i++)
   {
     for(j = 0; j < size-1; j++)
      {
         if(A[j] > A[j+1])
         {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
         }
      }
   }
}
```

Output:

```
Enter the values to be sorted:
10
9
7
8
After bubble sort:
7 8 9 10
```

INSERTION SORT

Insertion Sort

- Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Algorithm:

Step 1 - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.

Step 2: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.

Step 3: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Insertion Sort:

Example:

Consider the following unsorted list of elements...

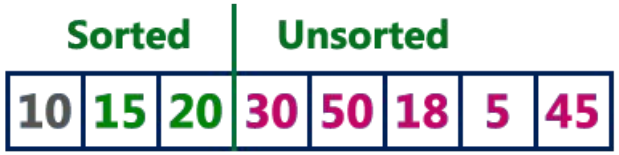
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Move the first element 15 from unsorted portion to sorted portion of the list.

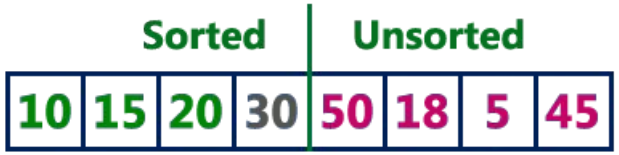
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

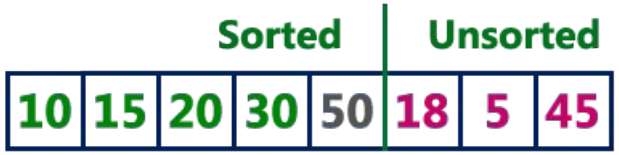
Insertion Sort: Example:



To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

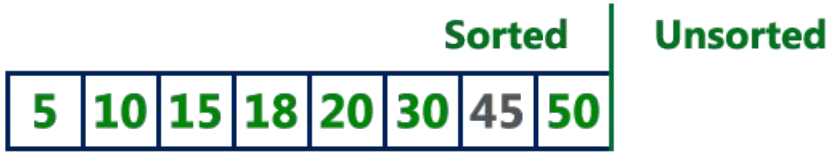


To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Insertion Sort:
Example:

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

Insertion Sort

Time Complexity of the Insertion Sort Algorithm:

To sort an unsorted list with **'n'** number of elements, we need to make ’n^2’ number of comparisons in the worst case.

If the list is already sorted then it requires **'n'** number of comparisons.

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(n^2)$

Insertion Sort: Program

```
#include<stdio.h>
#define size 4
int A[size];
void INSERTION_SORT(int A[]);
void main()
{
    int i;
    printf("Enter the values to be sorted:\n");
    for(i = 0; i <= size-1; i++)
    {
        scanf("%d", &A[i]);
    }
    INSERTION_SORT(A);
    printf("After insertion sort:\n");

    for(i = 0; i <= size-1; i++)
    {
        printf("%d ", A[i]);
    }
}
```

```
void INSERTION_SORT(int A[])
{
    int i, j, temp;
    for (i = 1; i < size; i++)
    {
        temp = A[i];
        j = i - 1;
        while ((A[j] > temp) && (j >= 0))
        {
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = temp;
    }
}
```

Output:

```
Enter the values to be sorted:
10
9
7
8
After insertion sort:
7 8 9 10
```

QUICK SORT

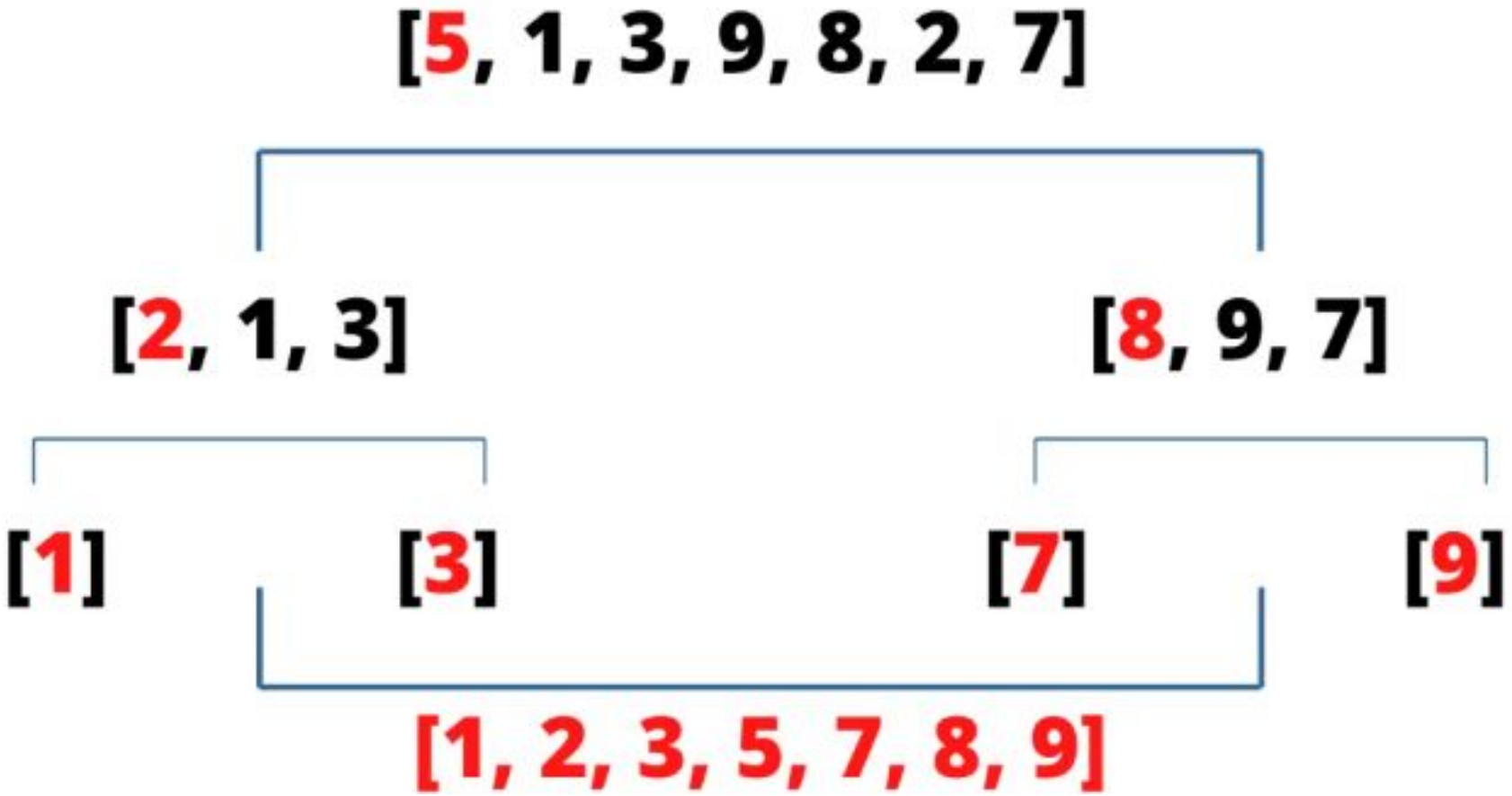
Quick Sort

- Quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively.
- It uses divide and conquer strategy.
- The partition of the list is performed based on the element called pivot.
- Pivot element is one of the elements in the list.
- The list is divided into two partitions such that "all elements to the left of the pivot are smaller than the pivot and all the elements to the right of pivot are greater than or equal to the pivot".

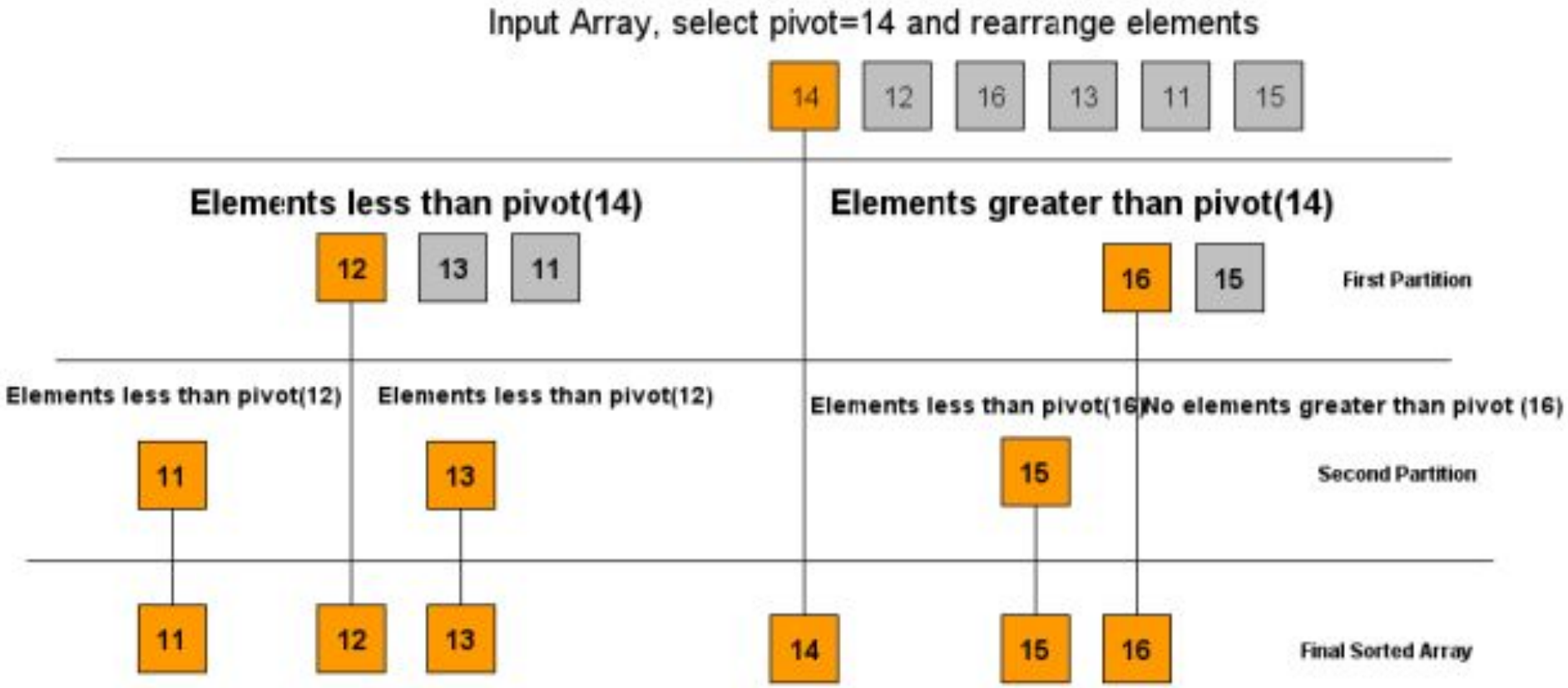
Algorithm:

Step 1: Consider the first element of the array as pivot (i.e., Element at first position in the array).
Step 2: Define two variables i and j. Set i and j to first and last elements of the array respectively.
Step 3: Increment i until array[i] >= pivot then stop.
Step 4: Decrement j until array[j] < pivot then stop.
Step 5: If i < j then exchange array[i] and array[j].
Step 6: Repeat steps 3, 4 & 5 until i > j.
Step 7: Exchange the pivot element with array[j] element.
Step 8: Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

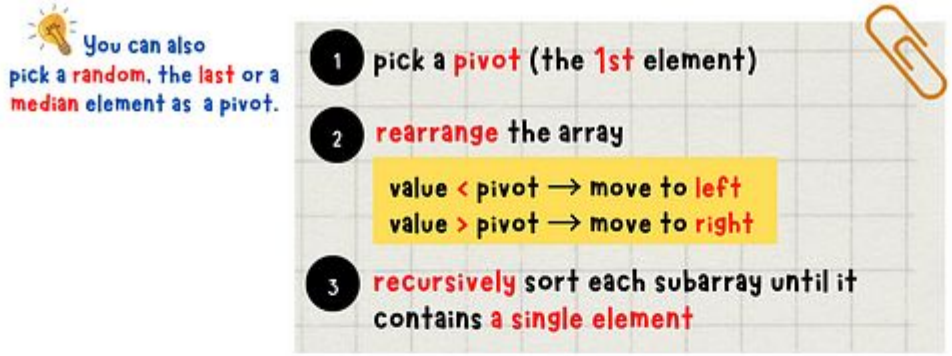
Quick Sort:
Example:

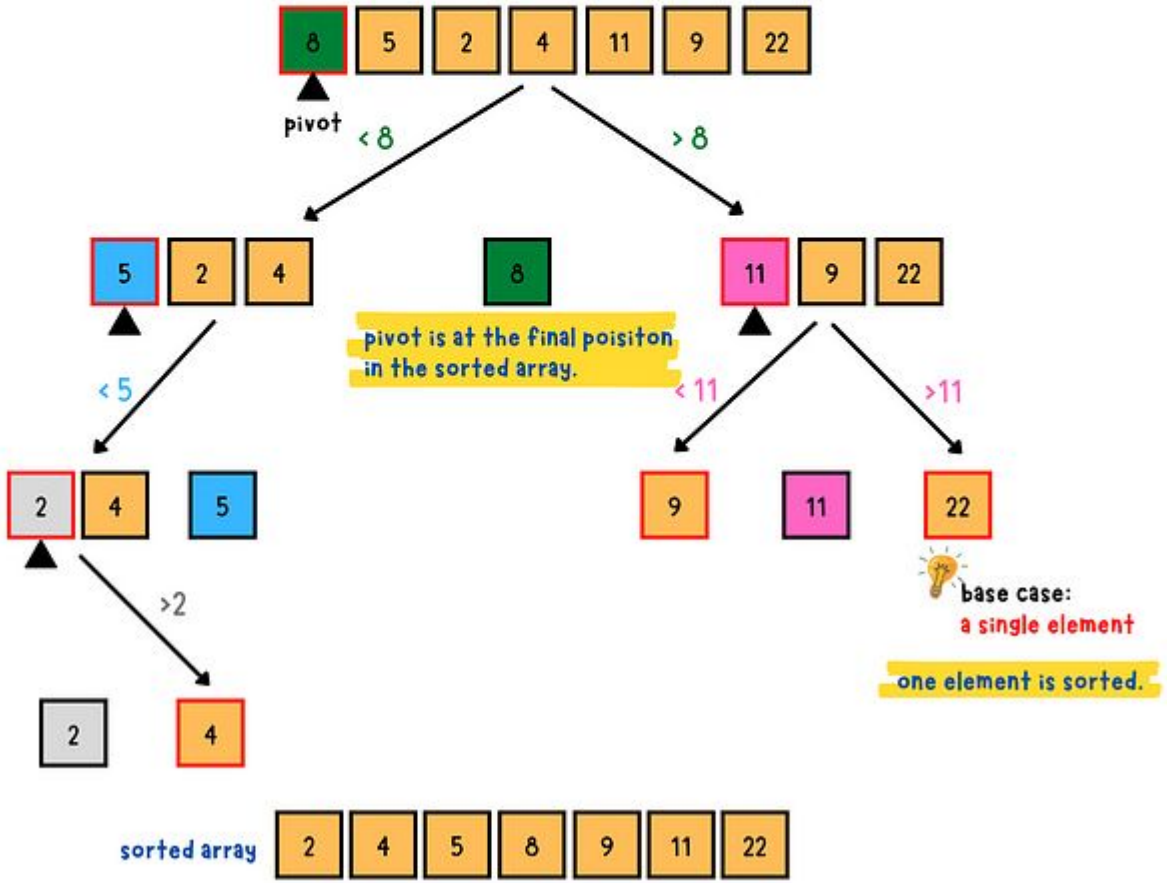


Quick Sort:
Example:



Quick Sort: Example:





Quick Sort: Example: Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

Quick Sort:

Example:

Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.

Quick Sort:
Example:

In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).

Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...

Quick Sort

Time Complexity of the Quick Sort Algorithm:

Worst Case: $O(N^2)$

Best Case: O(N log(N))

Average Case : O(N log(N))

Quick Sort: Program

```
#include<stdio.h>
#define size 4
int A[size];
void QUICK_SORT(int a[], int first, int last);

void main()
{
   int i;
   printf("Enter the values to be sorted:\n");
   for(i = 0; i <= size-1; i++)
   {
      scanf("%d", &A[i]);
   }
   QUICK_SORT(A, 0, size-1);
   printf("After Quick sort:\n");

   for(i = 0; i <= size-1; i++)
   {
      printf("%d ", A[i]);
   }
}
```

Output:

```
Enter the values to be sorted:
10
9
7
8
After Quick sort:
7 8 9 10
```

Quick Sort: Program

```
void QUICK_SORT(int A[10],int first,int last)
{
   int pivot,i,j,temp;
   if(first < last)
   {
      pivot = first;
      i = first;
      j = last;
      while(i < j)
      {
         while((A[pivot] > A[i]) && (i < last))
         {
            i++;
         }
         while(A[pivot] < A[j])
         {
            j--;
         }
         if(i <j)
         {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
         }
      }
      temp = A[pivot];
      A[pivot] = A[j];
      A[j] = temp;
      QUICK_SORT(A, first, j-1);
      QUICK_SORT(A,  j+1, last);
   }
}
```

MERGE SORT

Merge Sort

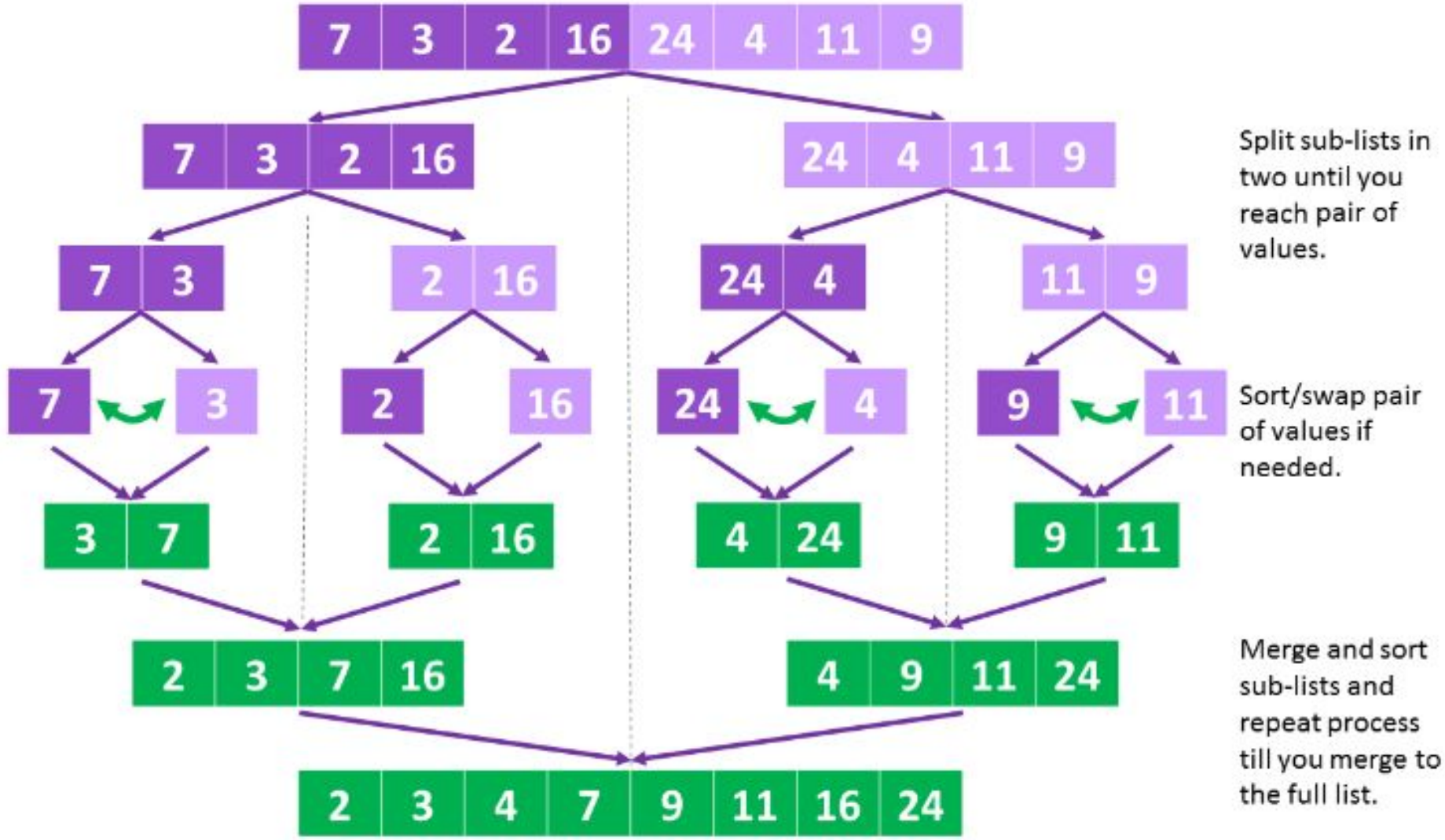
- It is based on a divide and conquer approach.
- Works by dividing an array into smaller sub arrays, sorting each sub array, and then merging the sorted sub arrays back together to form the final sorted array.
- It is a type of External sorting, where the data to be sorted is too large to fit into memory.

Algorithm:

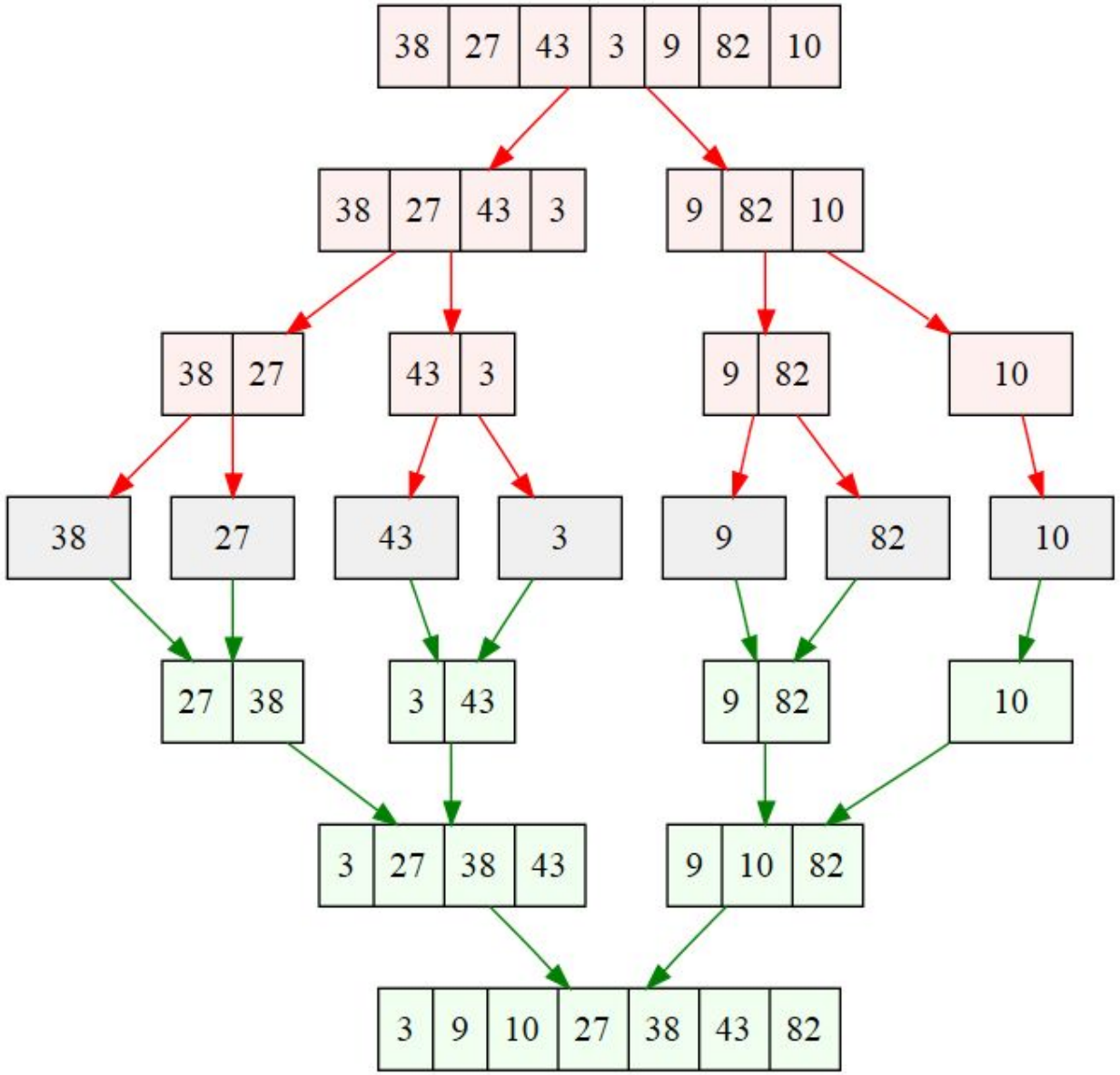
Step 1: Divide the unsorted list into n sub lists, each containing one element (a list of one element is considered sorted).

Step 2: Repeatedly merge sub lists to produce new sorted sub lists until there is only one sub list remaining. This will be the sorted list.

Merge Sort: Example:

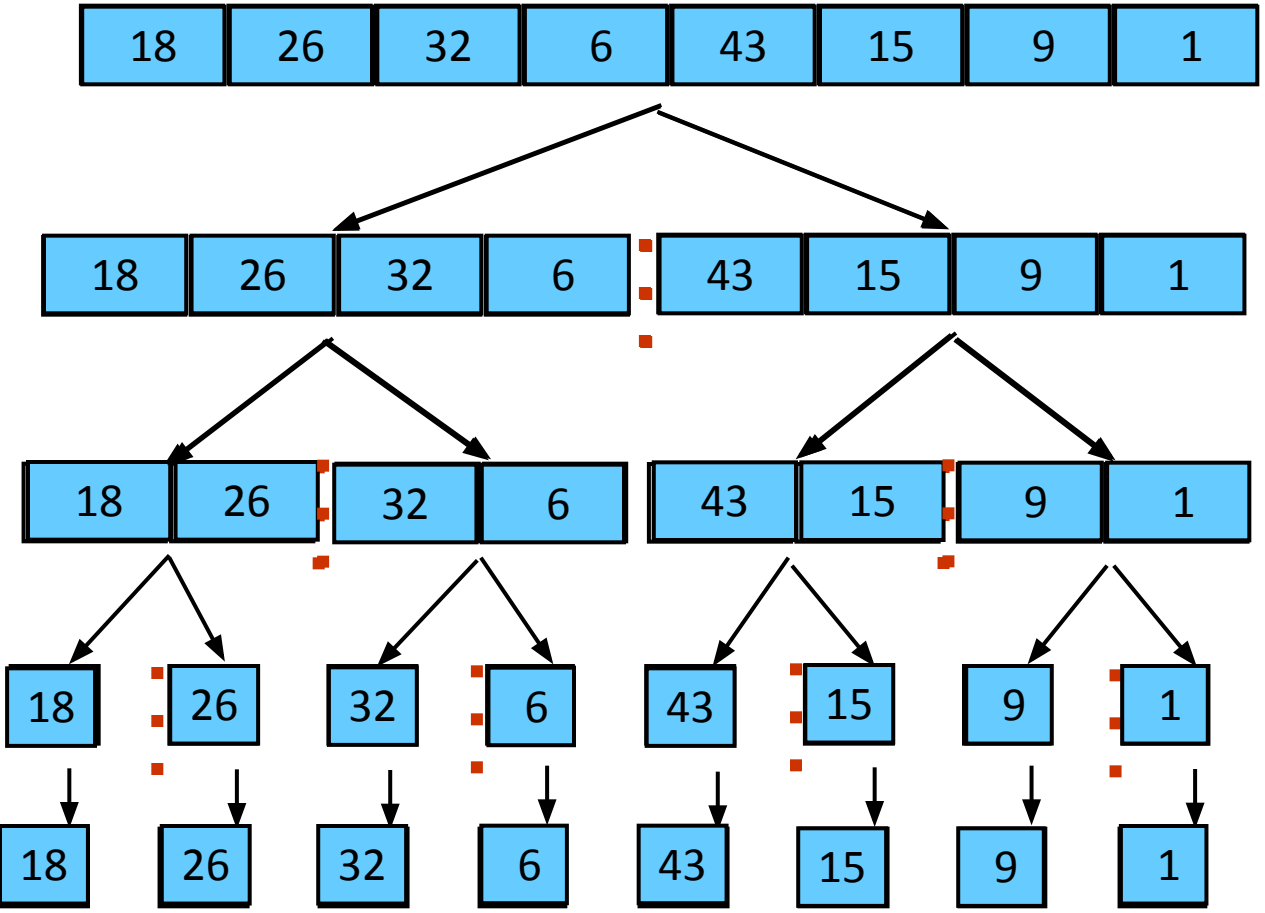


Merge Sort: Example:

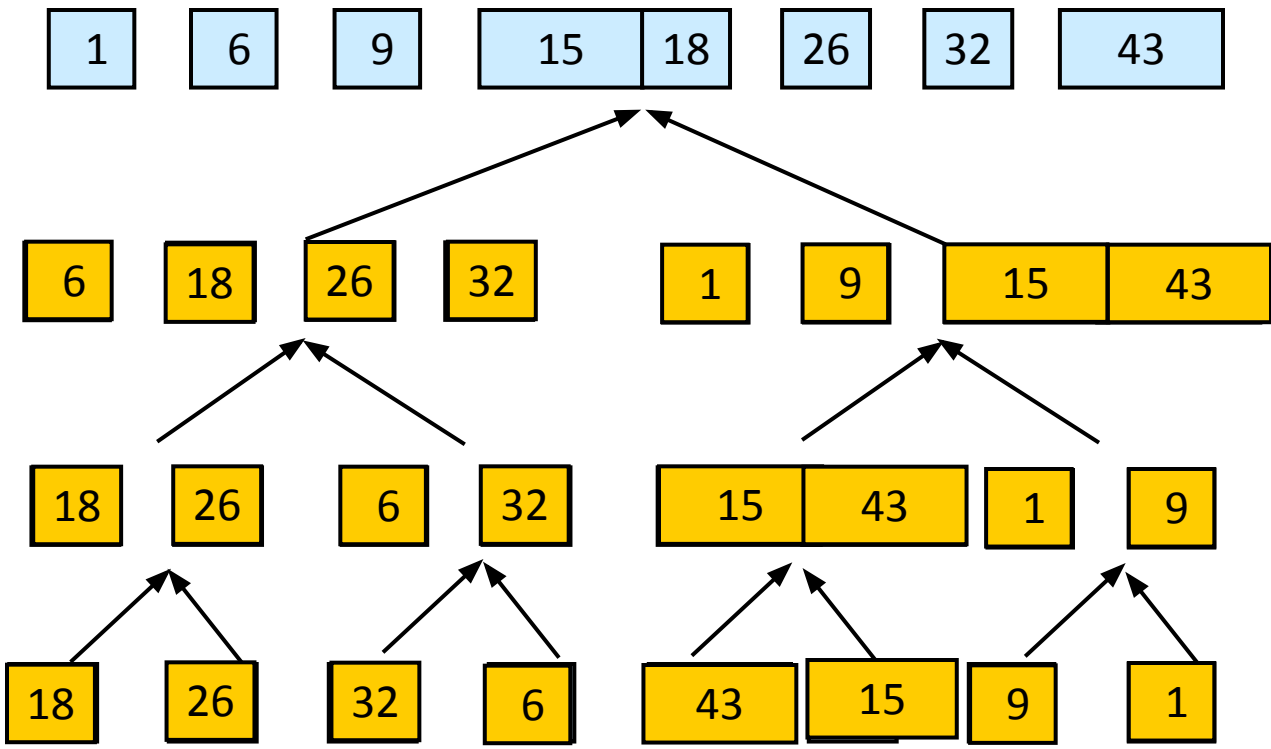


Merge Sort – Example

Original Sequence



Sorted Sequence



Merge Sort

Time Complexity of the Merge Sort Algorithm:

Worst Case / Best Case / Average Case : O(N log(N))

Merge Sort: Program

```
#include<stdio.h>
#define size 4
int A[size];
void MERGE_SORT(int A[], int le, int ri);
void merge(int A[], int le, int m, int ri);
void main()
{
   int i;
   printf("Enter the values to be sorted:\n");
   for(i = 0; i <= size-1; i++)
   {
      scanf("%d", &A[i]);
   }
   MERGE_SORT(A, 0, size-1);
   printf("After merge sort:\n");

   for(i = 0; i <= size-1; i++)
   {
      printf("%d ", A[i]);
   }
}
```

```
void MERGE_SORT(int A[], int left, int right)
{
   if (left < right)
   {
      int mid = (left + right) / 2;

      MERGE_SORT(A, left, mid);
      MERGE_SORT(A, mid + 1, right);

      merge(A, left, mid, right);
   }
}
```

Output:

```
Enter the values to be sorted:
10
9
7
8
After merge sort:
7 8 9 10
```

```
Enter the values to be sorted:
10
9
7
8
During merge l=0, m=0, r=1
During merge l=2, m=2, r=3
During merge l=0, m=1, r=3
After merge sort:
7 8 9 10
```

Merge Sort: Program

```
void merge(int A[], int left, int mid, int right)
{
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;
   printf("During merge l=%d, m=%d, r=%d", left, mid, right);
   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
   {
        L[i] = A[left + i];
   }
   for (j = 0; j < n2; j++)
   {
        R[j] = A[mid + 1 + j];
   }
   i = 0;
   j = 0;
   k = left;
   while (i < n1 && j < n2)
   {
        if (L[i] <= R[j])
        {
           A[k] = L[i];  i++;
        }
        else
        {
           A[k] = R[j];  j++;
        }
        k++;
   }
    while (i < n1)
   {
        A[k] = L[i];    i++;       k++;
   }
   while (j < n2)
   {
        A[k] = R[j];    j++;       k++;
   }
}
```

Data Structures (Module 5) – Searching

Searching

- Search is a process of finding a value in a list of values.

(OR)

- Searching is the process of locating given value position in a list of values.

Types:

- Linear Search Algorithm (Sequential Search Algorithm)
- Binary Search Algorithm

Linear Search Algorithm

- The search element (key element) is compared with element by element in the list.

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 Algorithm: Example:

search element 12

Step 1:

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

Both are not matching. So move to next element

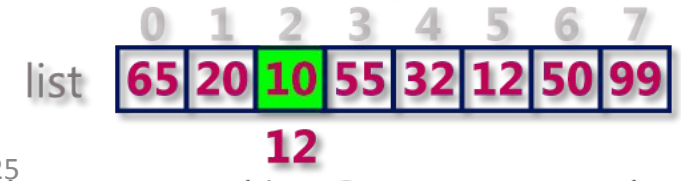
Step 2:

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

Both are not matching. So move to next element

Step 3:

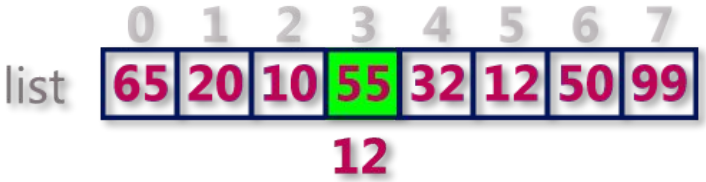
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

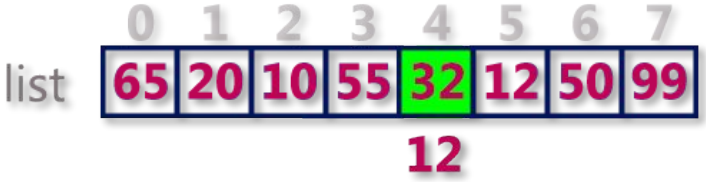
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

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



Both are not matching. So move to next element

Step 6:

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

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

Linear Search Algorithm

Time Complexity of the Linear Search Algorithm:

Worst Case: O(N)

Best Case: O(1)

Average Case : O(N)

Linear Search Algorithm: Program

```
#include<stdio.h>
#define size 4
int A[size];
int Linear_Search(int A[], int key);
void main()
{
   int i, key;
   printf("Enter the values:\n");
   for(i = 0; i <= size-1; i++)
   {
      scanf("%d", &A[i]);
   }
   printf("Enter the element to be searched:\n");
   scanf("%d", &key);
   int status = Linear_Search(A, key);
   if (status == -1)
   {   printf("Element is not found!!!");   }
    else
    {   printf("Element is found!!!");        }
}
```

```
int Linear_Search(int A[], int key)
{
     int i;
     for(i = 0; i <= size-1; i++)
     {
          if(A[i] == key)
          {
               return i;
          }
     }
     if(i == size)
     {
          return -1;
     }
}
```

Output:

```
Enter the values:
1
11
22
33
Enter the element to be searched:
44
Element is not found!!!
```

```
Enter the values:
1
11
22
33
Enter the element to be searched:
33
Element is found!!!
```

Binary Search Algorithm

- Binary search algorithm can be used with only a sorted list of elements.
- This search process starts comparing the search element with the middle element in the list.

Algorithm:

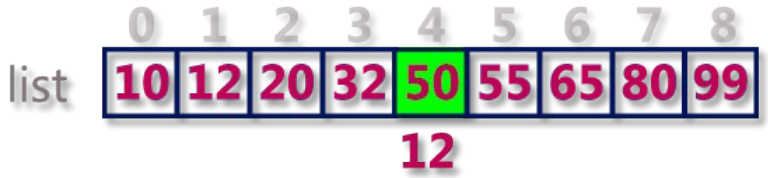
Step 1: Read the search element from the user.
Step 2: Find the middle element in the sorted list.
Step 3: Compare the search element with the middle element in the sorted list.
Step 4: If both are matched, then display "Given element is found!!!" and terminate the function.
Step 5: If both are not matched, 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 sub list 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 sub list of the middle element.
Step 8: Repeat the same process until we find the search element in the list or until sub list 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.

Binary Search Algorithm: Example:

search element 12

Step 1:

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



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

Step 2:

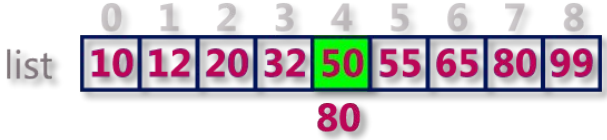
search element (12) is compared with middle element (12)

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)



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

Step 2:

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

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

Step 3:

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

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

Binary Search Algorithm

Time Complexity of the Linear Search Algorithm:

Worst Case: O(log N)

Best Case: O(1)

Average Case : O(log N)

Binary Search Algorithm: Program

```
#include<stdio.h>
#define size 4
int A[size];
int Binary_Search(int A[], int key);
void main()
{
   int i, key;
   printf("Enter the values in ASCENDING ORDER:\n");
   for(i = 0; i <= size-1; i++)
   {
      scanf("%d", &A[i]);
   }
   printf("Enter the element to be searched:\n");
   scanf("%d", &key);
   int status = Binary_Search(A, key);
   if (status == -1)
   {   printf("Element is not found!!!");   }
    else
    {   printf("Element is found!!!");        }
}
```

```
int Binary_Search(int A[], int key)
{
   int first = 0;
   int last = size - 1;
   int middle = (first+last)/2;
   while (first <= last)
   {
      if (A[middle] < key)
      {   first = middle + 1;       }
      else if (A[middle] > key)
      {   last = middle - 1;        }
      else
      {   return middle;            }
      middle = (first + last)/2;
   }
   if (first > last)
   {   return -1;  }
}
```

Output:

```
Enter the values in ASCENDING ORDER:
1
11
22
33
Enter the element to be searched:
33
Element is found!!!
```

```
Enter the values in ASCENDING ORDER:
1
11
22
33
Enter the element to be searched:
44
Element is not found!!!
```

Comparison

- Linear search
  - Small arrays
  - Unsorted arrays
- Binary search
  - Large arrays
  - Sorted arrays

Data Structures (Module 5) – Hashing

Hashing

- Hashing:
  - Hash table
  - Hash functions
  - Collision Resolution Techniques

Hashing

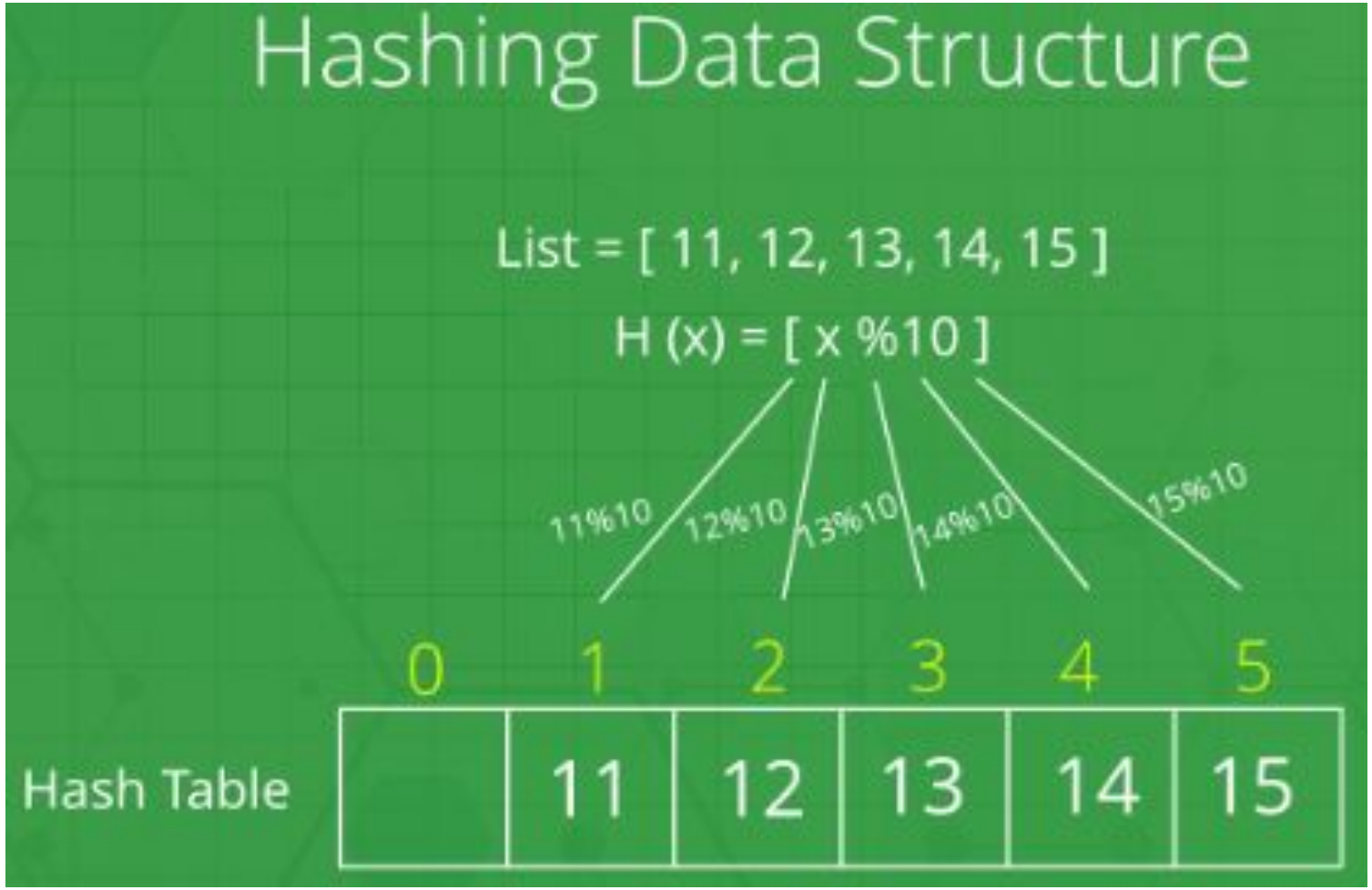
- Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.

(OR)

- Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function.
- It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.
- Examples of how hashing is used in our lives:
  - In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
  - In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

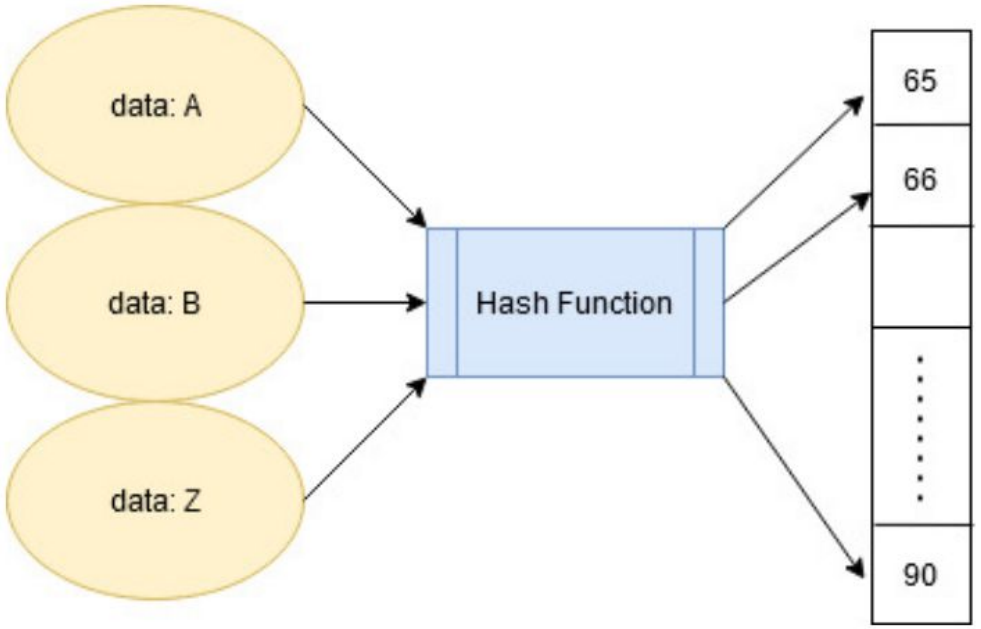
Hashing

- Let a hash function H(x) maps the value x at the index x%10 in an Array.
- For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



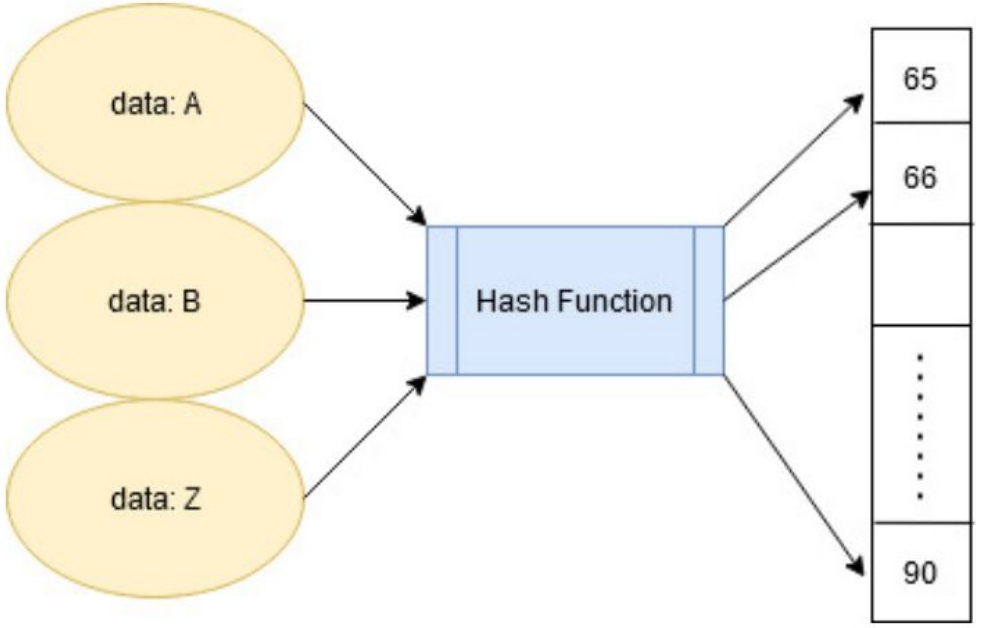
Hashing

- Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use hashing.
- In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key).
- By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.



Hashing

- **Hash table:** A data structure where the data is stored based upon its hashed key which is obtained using a hashing function.
- **Hash function:** A function outputs a value mapped to a fixed range for a given data. A hash table leverage the hash function to efficiently map data such that it can be retrieved and updated quickly.
- Assume S = {s1, s2, s3, ..., sn} to be a set of objects that we wish to store into a map of size N, so we use a hash function H, such that for all s belonging to S; H(s) -> x, where x is guaranteed to lie in the range [1,N]
- **Perfect Hash function:** A hash function that maps each item into a unique slot (no collisions).



Hashing

- Hashing is implemented in two steps:
  1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
  2. The element is stored in the hash table where it can be quickly retrieved using hashed key.
     - hash = hashfunc(key)
     - index = hash % array_size
     - In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Hashing

- **Hash function:**
- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:
    1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
    2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
    3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.
- Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Hashing

- **Hash Collisions:**
  - As per the Pigeonhole principle if the set of objects we intend to store within our hash table is larger than the size of our hash table we are bound to have two or more different objects having the same hash value; a hash collision.
  - Even if the size of the hash table is large enough to accommodate all the objects finding a hash function which generates a unique hash for each object in the hash table is a difficult task.
  - Collisions are bound to occur (unless we find a perfect hash function, which in most of the cases is hard to find) but can be significantly reduced with the help of various collision resolution techniques.
- **Collision resolution techniques:**
- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
  - Liner Probing
  - Quadratic probing
  - Double hashing

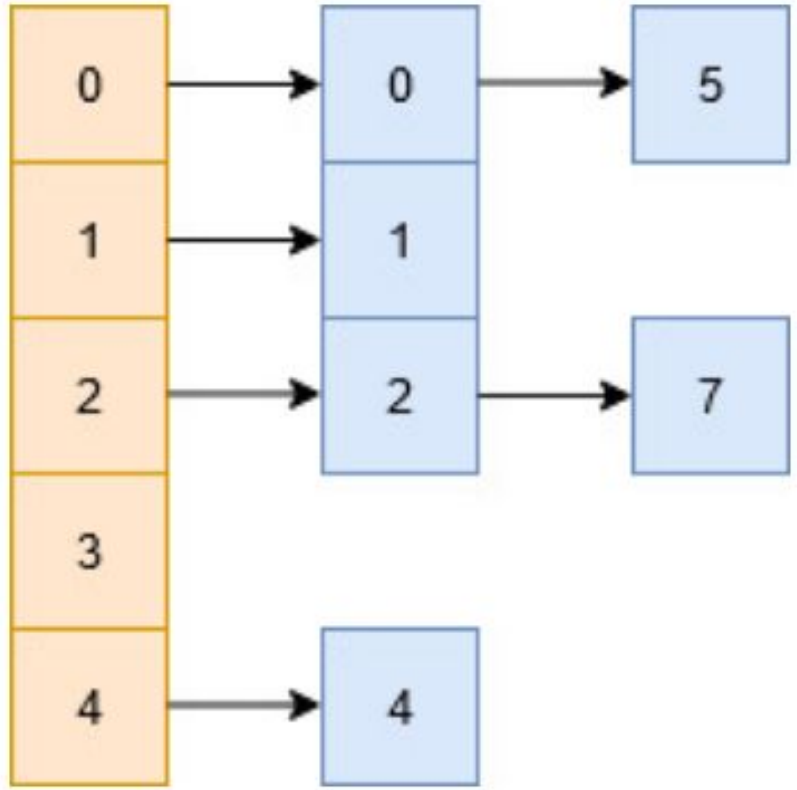
Hashing

- **Open Hashing (Separate chaining)**
- Collisions are resolved using a list of elements to store objects with the same key together.
- Suppose you wish to store a **set of numbers = {0,1,2,4,5,7}** into a **hash table of size 5**.
- Now, assume that we have a **hash function H**, **H(x) = x%5**
- So, if we were to map the given data with the given hash function we'll get the corresponding values.
  - H(0)-> 0%5 = 0
  - H(1)-> 1%5 = 1
  - H(2)-> 2%5 = 2
  - H(4)-> 4%5 = 4
  - H(5)-> 5%5 = 0
  - H(7)-> 7%5 = 2
- Clearly **0 and 5**, as well as **2 and 7** will have the same hash value, and in this case we will simply append the colliding values to a list being pointed by their hash keys.

Hashing

- **Open Hashing (Separate chaining)**

```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
H(7)-> 7%5 = 2
```

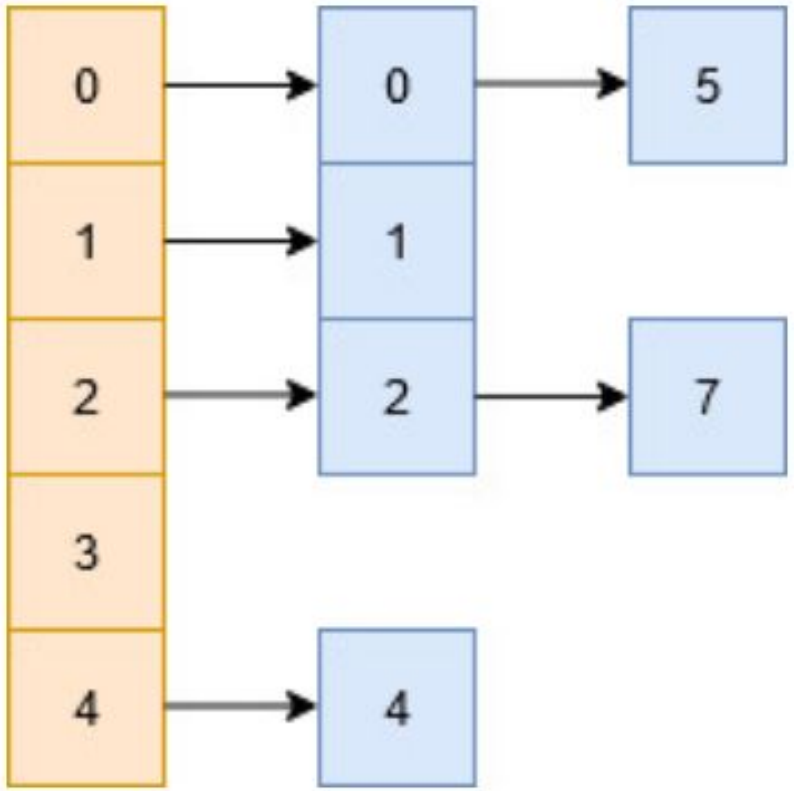


- Clearly **0 and 5**, as well as **2 and 7** will have the same hash value, and in this case we will simply append the colliding values to a list being pointed by their hash keys.

Hashing

```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
H(7)-> 7%5 = 2
```

- **Open Hashing (Separate chaining)**
- This is an easy way to implement hashing but it has its own **demerits**.
- The lookups/inserts/updates can become linear [O(N)] instead of constant time [O(1)] if the hash function has too many collisions.
- It doesn't account for any empty slots which can be leveraged for more efficient storage and lookups.
- Ideally we require a good hash function to guarantee even distribution of the values.
- Say, for a load factor λ=number of objects stored in table/size of the table (can be >1) a good hash function would guarantee that the maximum length of list associated with each key is close to the load factor.



Hashing

- **Closed Hashing (Open Addressing)**
- This collision resolution technique requires a hash table with fixed and known size. During insertion, if a collision is encountered, alternative cells are tried until an empty bucket is found.
- These techniques require the size of the hash table to be larger than the number of objects to be stored (something with a load factor < 1 is ideal).
- There are various methods to find these empty buckets:
    1. Liner Probing
    2. Quadratic probing
    3. Double hashing

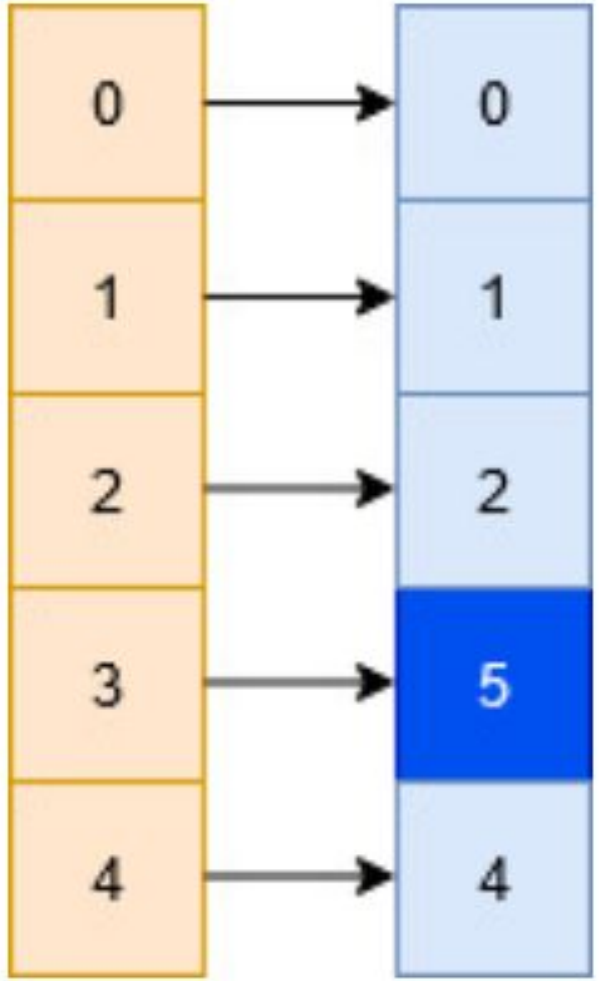
Hashing

- **Closed Hashing (Open Addressing): Linear Probing**
- we take a fixed sized hash table and every time we face a hash collision we linearly traverse the table in a cyclic manner to find the next empty slot.
- Assume a scenario where we intend to store the following **set of numbers = {0,1,2,4,5}** into a hash table of size 5 with the help of the following **hash function H, such that H(x) = x%5**
- So, if we were to map the given data with the given hash function we'll get the corresponding values.
  - H(0)-> 0%5 = 0
  - H(1)-> 1%5 = 1
  - H(2)-> 2%5 = 2
  - H(4)-> 4%5 = 4
  - H(5)-> 5%5 = 0

Hashing

```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
```

- **Closed Hashing (Open Addressing): Linear Probing**
- In this case we see a collision of two terms (0 & 5).
- In this situation we move linearly down the table to find the first empty slot.
- Note that this linear traversal is cyclic in nature, i.e. in the event we exhaust the last element during the search we start again from the beginning until the initial key is reached.



Hashing

- **Closed Hashing (Open Addressing): Linear Probing**
- In this case our hash function can be considered as this:

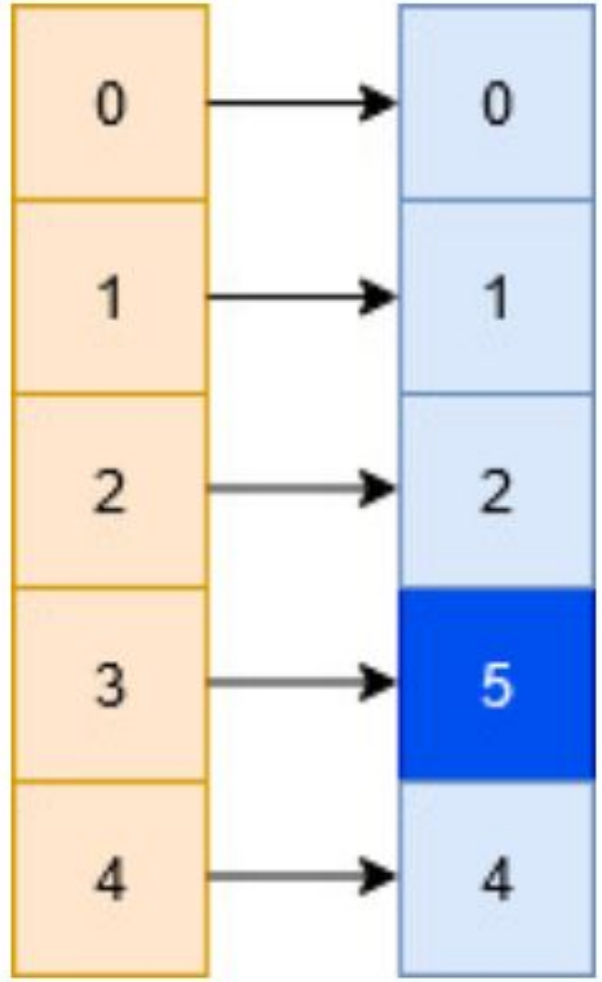
H(x, i) = (H(x) + i)%N = ((x%N) + i)%N

- H(0,0) = ((0%5)+0)%5 = 0
- H(1,0) = ((1%5)+0)%5 =1%5 = 1
- H(2,0) = ((2%5)+0)%5 =2%5 = 2
- H(4,0) = ((4%5)+0)%5 = 4%5 = 4
- H(5,0) = ((5%5)+0)%5 =0%5 = 0 (Collision)
- H(5,1) = ((5%5)+1)%5 =1%5 = 1 (Collision)
- H(5,2) = ((5%5)+2)%5 =2%5 = 2 (Collision)
- H(5,3) = ((5%5)+3)%5 =3%5 = 3 (Empty, place 5 at 3rd index)

where N is the size of the table and i represents the linearly increasing variable which starts from 0 (until empty bucket is found).

- Despite being easy to compute, implement and deliver best cache performance, this suffers from the problem of clustering (many consecutive elements get grouped together, which eventually reduces the efficiency of finding elements or empty buckets).

```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
```



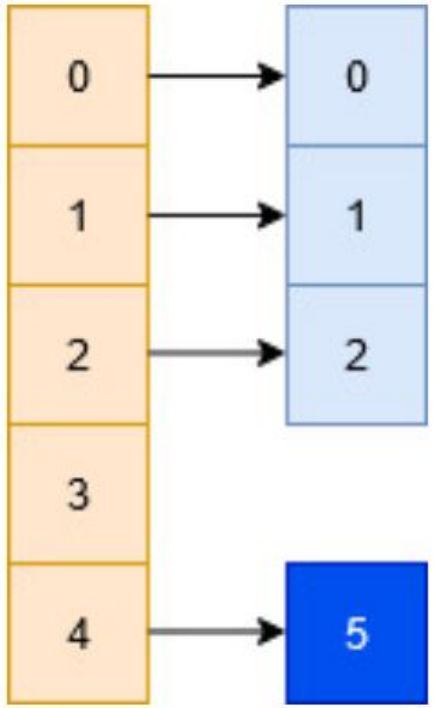
Hashing

- **Closed Hashing (Open Addressing): Quadratic Probing**
- A simple expression of Q would be Q(i) = i^2, in which case the hash function looks something like this: **H(x, i) = (H(x) + i^2)%N**
- In general, H(x, i) = (H(x) + ((c1/*i^2 + c2/*i + c3)))%N, for some choice of constants c1, c2, and c3.
- Despite resolving the problem of clustering significantly it may be the case that in some situations this technique does not find any available bucket, unlike linear probing which always finds an empty bucket.
- We can get good results from quadratic probing with the right combination of probing function and hash table size which will guarantee that we will visit as many slots in the table as possible.
- In particular, if the hash table's size is a prime number and the probing function is H(x, i) = i^2, then at least 50% of the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will eventually be found.

Hashing

- **Closed Hashing (Open Addressing): Quadratic Probing**
- Alternatively, if the hash table size is a power of two and the probing function is H(x, i) = (i^2 + i)/2, then every slot in the table will be visited by the probing function.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that H(x, i) = (x%5 + i^2)%5.

- $H(0,0) = ((0\%5)+0^2)\%5 = 0\%5 = 0$
- $H(1,0) = ((1\%5)+0^2)\%5 = 1\%5 = 1$
- $H(2,0) = ((2\%5)+0^2)\%5 = 2\%5 = 2$
- $H(5,0) = ((5\%5)+0^2)\%5 = 0\%5 = 0$ (Collision)
- $H(5,1) = ((5\%5)+1^2)\%5 = 1\%5 = 1$ (Collision)
- $H(5,2) = ((5\%5)+2^2)\%5 = 4\%5 = 4$ (Empty, place 5 at 4th index)



Hashing

- **Closed Hashing (Open Addressing): Double Hashing**
- In the event of a collision we use an another hashing function with the key value as an input to find where in the open addressing scheme the data should actually be placed at.
- In this case we use two hashing functions, such that the final hashing function looks like: H(x, i) = (H1(x) + i*H2(x))%N
- Typically for H1(x) = x%N a good H2 is H2(x) = P - (x%P), where P is a prime number smaller than N.
- A good H2 is a function which never evaluates to zero and ensures that all the cells of a table are effectively traversed.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that
  - **H(x, i) = (H1(x) + i*H2(x))%5**
  - **H1(x) = x%5 and H2(x) = P - (x%P),** where P = 3 (3 is a prime smaller than 5)

Hashing

- **Closed Hashing (Open Addressing): Double Hashing**

H(x, i) = (H1(x) + i*H2(x))%5

H1(x) = x%5 and H2(x) = P - (x%P),

H(0, 0) = ((0%5)+0*(3-(0%3)))%5 = 0%5 = 0

H(1,0) = ((1%5)+0*(3-(1%3)))%5 = 1%5 = 1

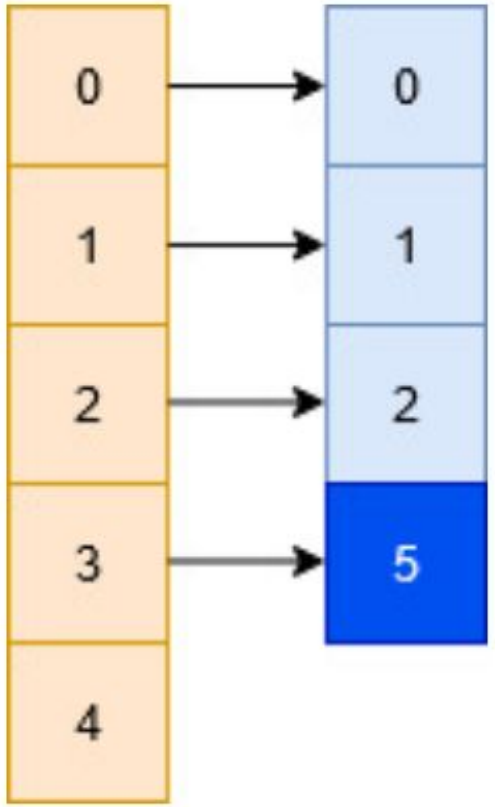
H(2,0) = ((2%5)+0*(3-(2%3)))%5 = 2%5 = 2

H(5,0) = ((5%5)+0*(3-(5%3)))%5 = 0%5 = 0 (Collision)

H(5,1) = ((5%5)+1*(3-(5%3)))%5 = 1%5 = 1 (Collision)

H(5,2) = ((5%5)+2*(3-(5%3)))%5 = 2%5 = 2 (Collision)

H(5,3) = ((5%5)+3*(3-(5%3)))%5 = 3%5 = 3 (Empty, place 5 at 3rd index)



Hashing

```
#include<stdio.h>

#define size 7

int arr[size];

void initialize();
void insert(int value);
void del(int value);
void search(int value);
void print();
```

```
void main()
{
    initialize();
    insert(10); //key = 10 % 7 ==> 3
    insert(4);  //key = 4 % 7  ==> 4
    insert(3);  //key = 3 % 7  ==> 3 (collision)

    printf("Hash table\n");              print();   printf("\n");

    printf("Deleting value 10..\n");         del(10);

    printf("After the deletion hash table\n");   print();   printf("\n");

    printf("Deleting value 5..\n");          del(5);

    printf("After the deletion hash table\n");   print();   printf("\n");

    printf("Searching value 4..\n");             search(4);
    printf("Searching value 10..\n");        search(10);
}
```

```
10 inserted at arr[3]
4 inserted at arr[4]
Collision : arr[3] has element 10 already!
Unable to insert 3
Hash table
arr[0] = -1
arr[1] = -1
arr[2] = -1
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 10..
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = -1
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 5..
5 not present in the hash table
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = -1
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Searching value 4..
Search Found
Searching value 10..
Search Not Found
```

Hashing

```
void initialize()
{
    int i;
    for(i = 0; i < size; i++)
    {    arr[i] = -1;        }
}


void print()
{
    int i;
    for(i = 0; i < size; i++)
    {    printf("arr[%d] = %d\n", i, arr[i]);}
}
```

```
void insert(int value)
{
    int key = value % size;
    if(arr[key] == -1)
    {
        arr[key] = value;
        printf("%d inserted at arr[%d]\n", value, key);
    }
    else
    {
        printf("Collision: arr[%d] has element %d already!\n", key, arr[key]);
        printf("Unable to insert %d\n", value);
    }
}
void del(int value)
{
    int key = value % size;
    if(arr[key] == value)
    {    arr[key] = -1;    }
    else
     {   printf("%d not present in the hash table\n", value);   }
}
```

Thank you!