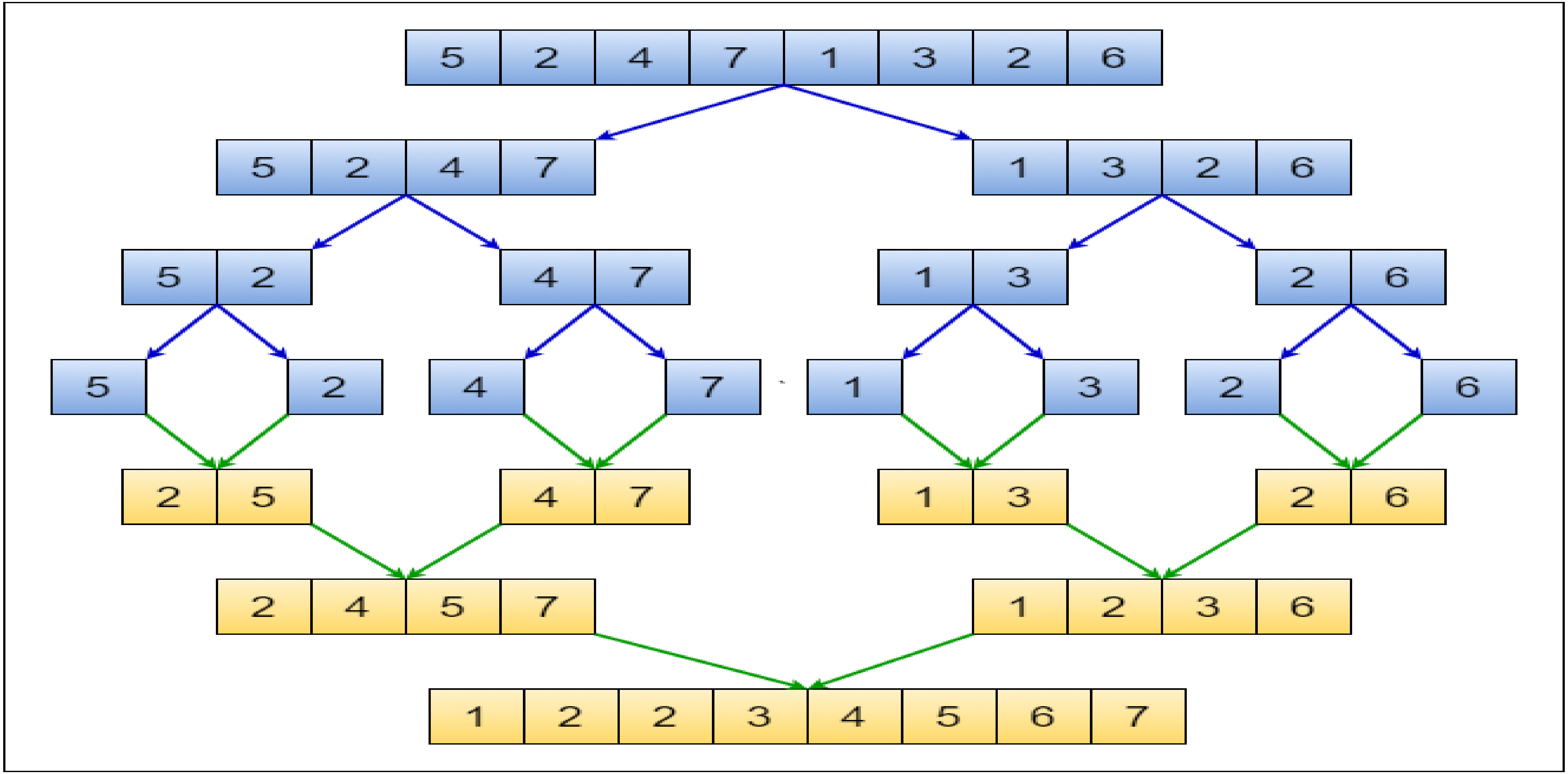
# Algorithms & Data Structure

Kiran Waghmare

# Example


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

# How MergeSort Algorithm Works Internally

1. Divide the array into two parts

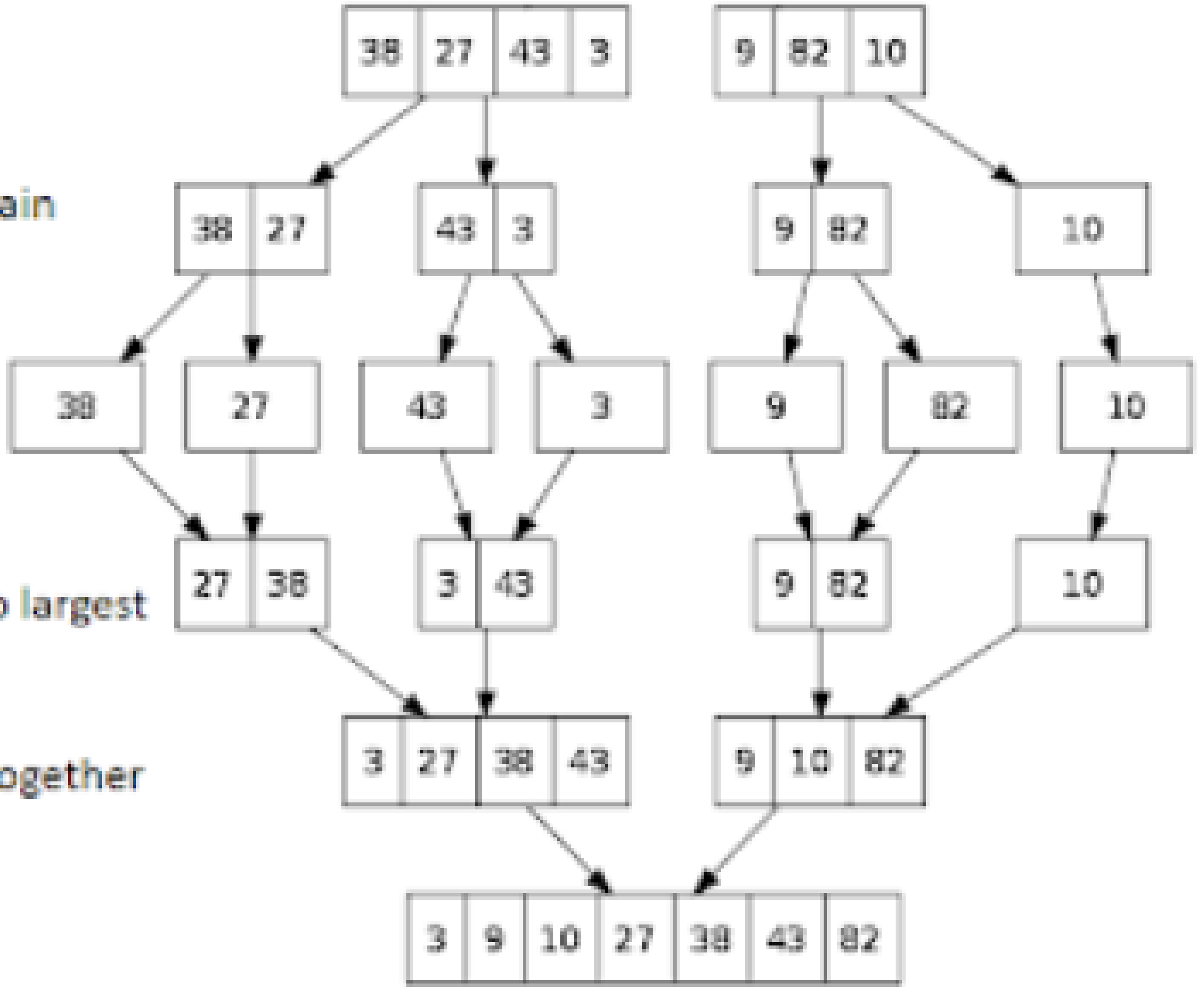
2. Divide the array into two parts again

3. Break each element into single parts

4. Sort the elements from smallest to largest

5. Merge the divided sorted arrays together

6. The array has been sorted



Merge Sort:

-------------

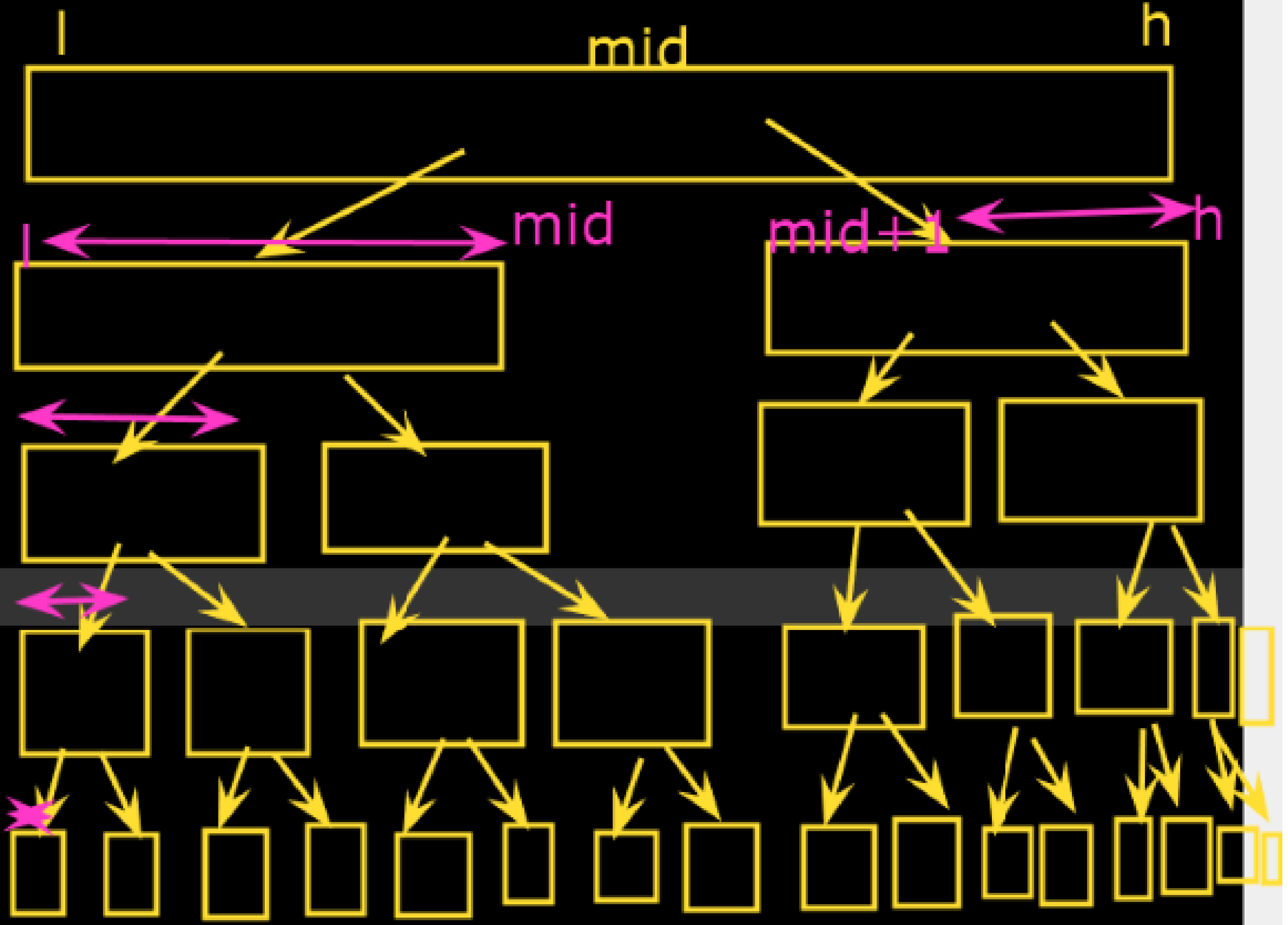
-Divide & Conquer

-Recursion

Algorithm:

```
mergesort(l,h)
{
    if(l<h)
    {
        mid=l+h/2;
        mergesort(l,mid);
        mergesort(mid+1,h);

    }
}
```



```
 Merge sort (A, p, r)
{ if (p < r)                          // check for base case
   q = [ (p + r)/2 ]                  // divide step
 Merge sort (A, p, q)                 // conquer step
 Merge sort (A, q+1, r)               // conquer step
 Merge sort (A, p, q, r)              // conquer step
}
```

MERGE$(A, p, q, r)$

```
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

# Sorting

- **Insertion sort**
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- **Bubble Sort**
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

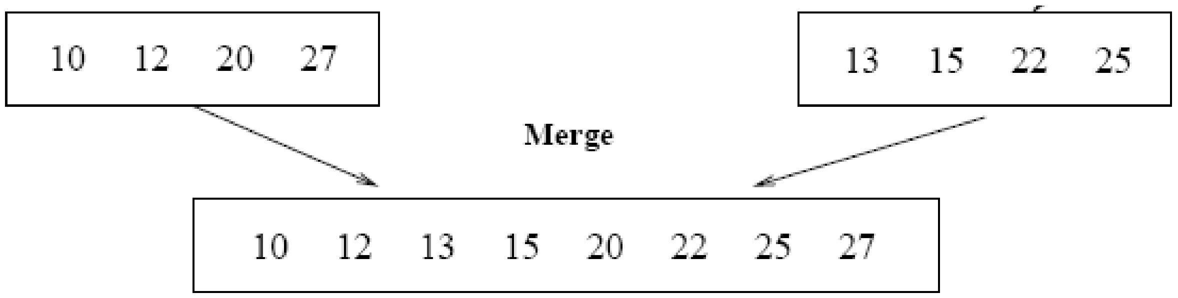
# Sorting

- **Selection sort**
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- **Merge Sort**
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: Let's see!!

# Running Time of Merge (assume last for loop)

- **Initialization (copying into temporary arrays):**
  - $\Theta(n_1 + n_2) = \Theta(n)$
- **Adding the elements to the final array:**
  - $n$ iterations, each taking constant time $\Rightarrow \Theta(n)$
- **Total time for Merge:**
  - $\Theta(n)$



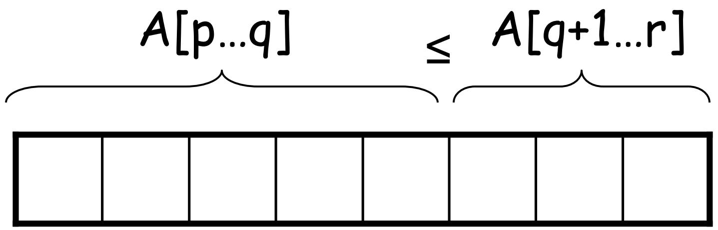
# Merge Sort - Discussion

- **Running time insensitive of the input**

- **Advantages:**
  - Guaranteed to run in $\Theta(nlgn)$

- **Disadvantage**
  - Requires extra space $\approx$N

# Quicksort




- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

# QUICKSORT

*Alg.:* QUICKSORT($A$, p, r)

if p < r

then $q \leftarrow$ PARTITION($A$, p, r)

QUICKSORT ($A$, p, $q$)

QUICKSORT ($A$, $q$+1, r)

Initially: p=1, r=n

Recurrence:

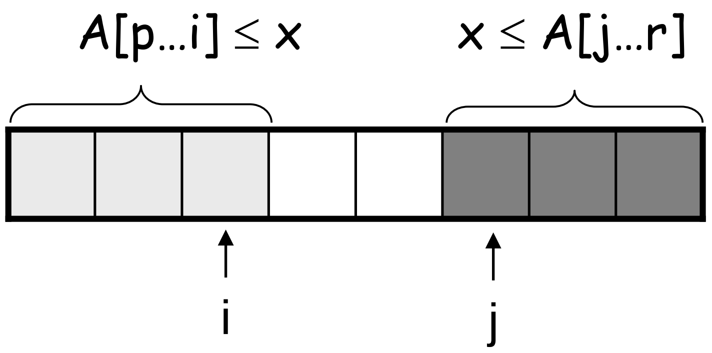
$T(n) = T(q) + T(n - q) + f(n)$   ($f(n)$ depends on PARTITION())

# Partitioning the Array

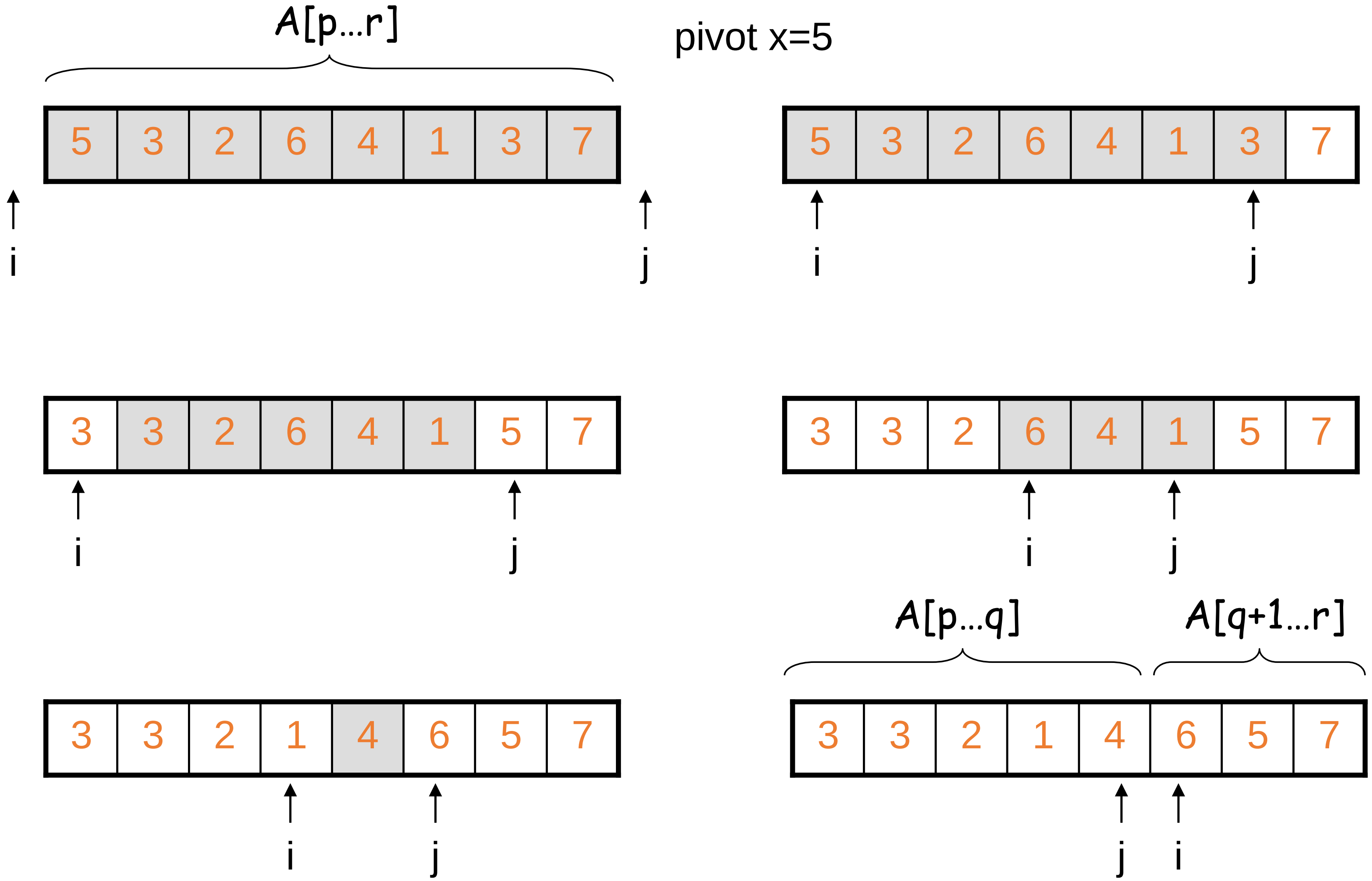
- **Choosing PARTITION()**
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- **Hoare partition**
- **Select a pivot element x around which to partition**
  - Grows two regions

    $A[p \ldots i] \leq x$

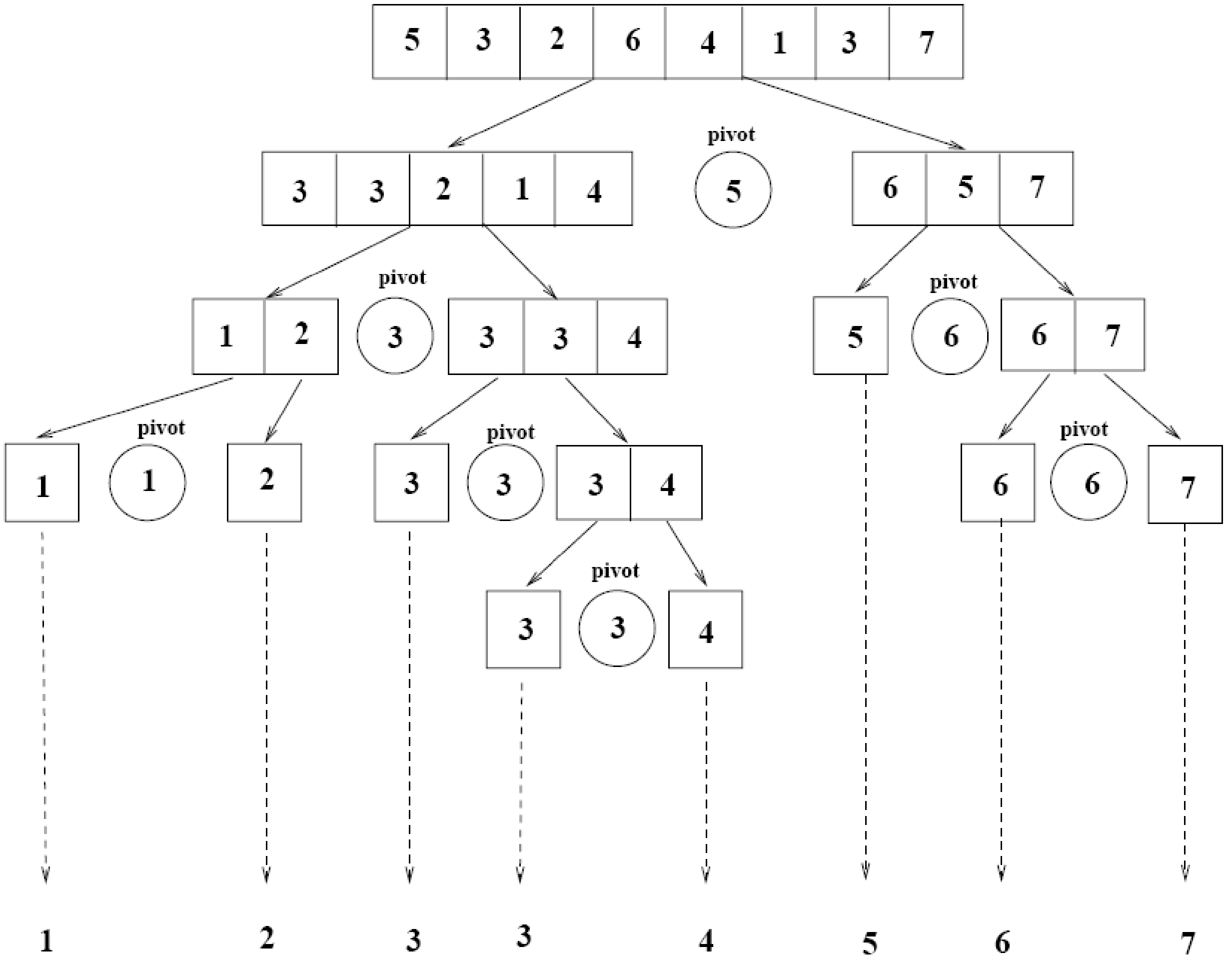
    $x \leq A[j \ldots r]$



# Example


A[p…r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 1 4 6 5 7
i
j
A[p…q]
A[q+1…r]
3 3 2 1 4 6 5 7
j
i

# Example


5 3 2 6 4 1 3 7
3 3 2 1 4
pivot
5
6 5 7
1 2
pivot
3
3 3 4
5
pivot
6
6 7
1
pivot
1
2
3
pivot
3
3 4
6
pivot
6
7
3
pivot
3
4
1 2 3 3 4 5 6 7

Example:

i> pivot
j<pivot

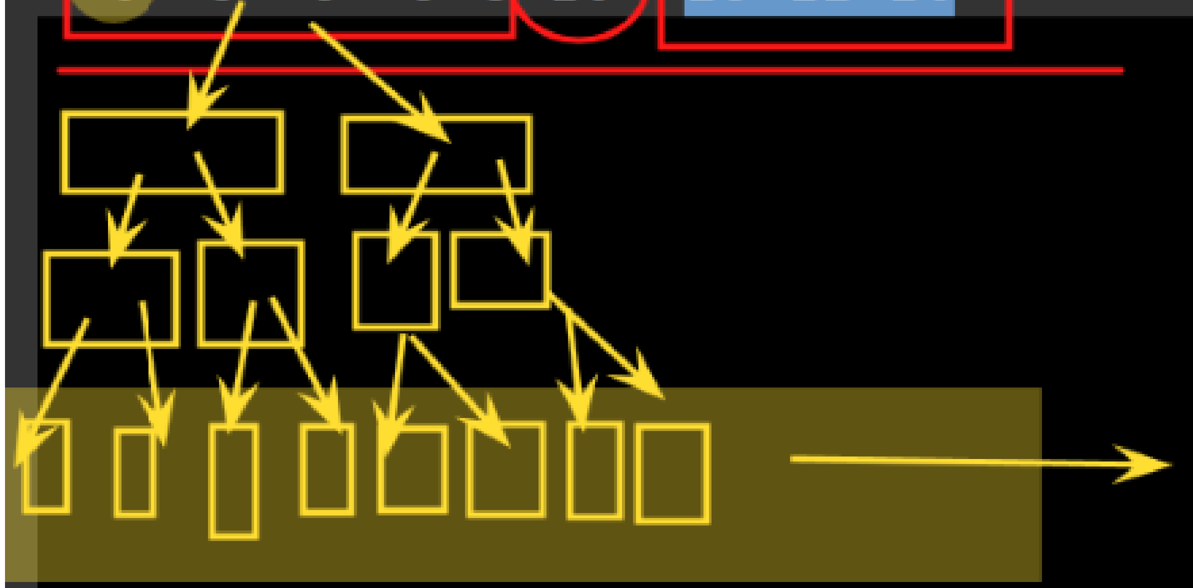
10 16 8 12 15 6 3 9 5 #

10 5 8 12 15 6 3 9 16 #

10 5 8 9 15 6 3 12 16 #

10 5 8 9 3 6 15 12 16 #

6 5 8 9 3 10 15 12 16

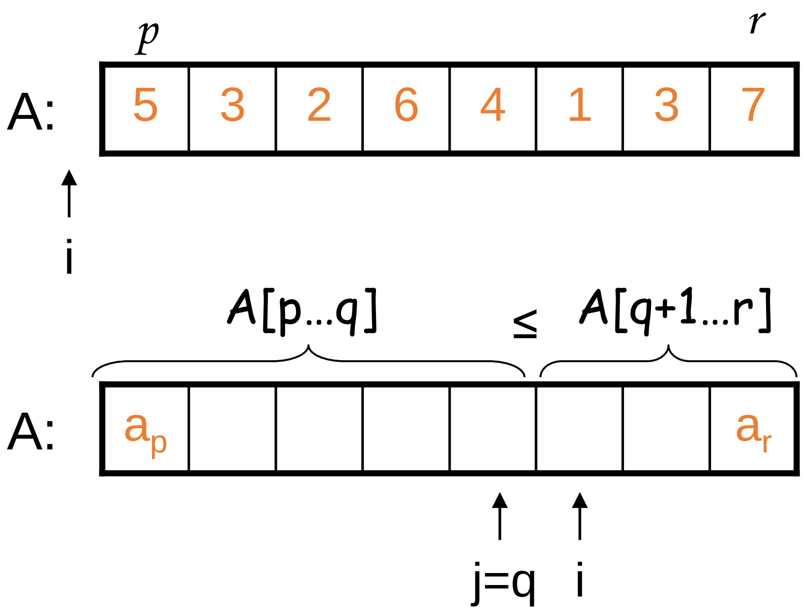


sorted arry

# Partitioning the Array

*Alg.* **PARTITION (A, p, r)**

```
x ← A[p]
i ← p – 1
j ← r + 1
while TRUE
        do repeat j ← j – 1
              until A[j] ≤ x
        do  repeat i ← i + 1
              until A[i] ≥ x
         if i < j
              then exchange A[i] ↔ A[j]
         else return j
```




Each element is visited once!

Running time: $\Theta(n)$

$n = r - p + 1$

# Recurrence

*Alg.:* QUICKSORT($A$, p, r)

Initially: p=1, r=n

if p < r

then $q \leftarrow$ PARTITION($A$, p, r)

QUICKSORT ($A$, p, $q$)

QUICKSORT ($A$, $q$+1, r)

Recurrence:

$$T(n) = T(q) + T(n - q) + n$$

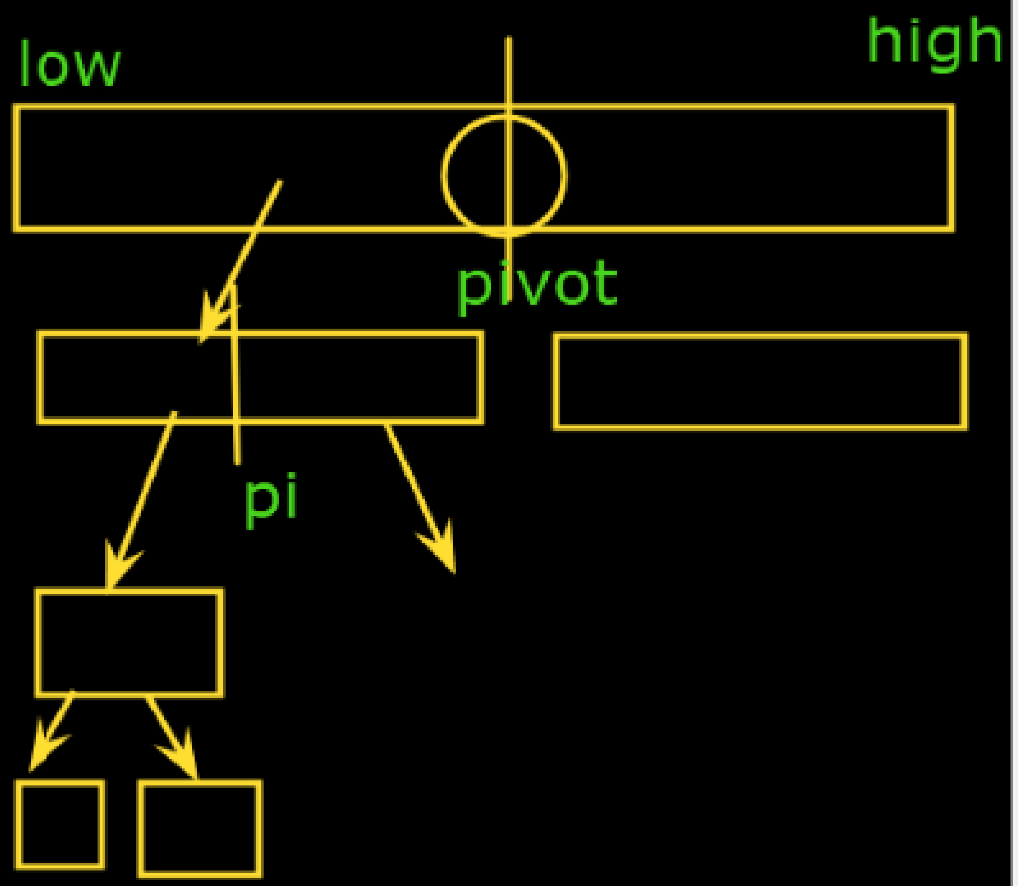
```
        j++;
        k++;
    }
    }

}
void  Quicksort(int a1[],int low, int high)
{
    if(low < high)
    {
        int pi = partition(a1,low,high);
        Quicksort(a1,low,pi-1);
        Quicksort(a1,pi+1,high);

    }
}



 int partition(int a1[],int low, int high)
 {
    int pivot = a1[high]
    int i =(low-1);
```



```
B: 5 9 14 17 31 35 65 78  89 92

C: 2 5 6 9  9 14 15 17 20 24 27 31 35 65 78  89 92

Quick sort:
-----------
```

< pivot

>pivot

```
80 90 30 50 70 10

10 205 305 401 50
```

Best case

```
50 40 30 20 10
```

Worst case

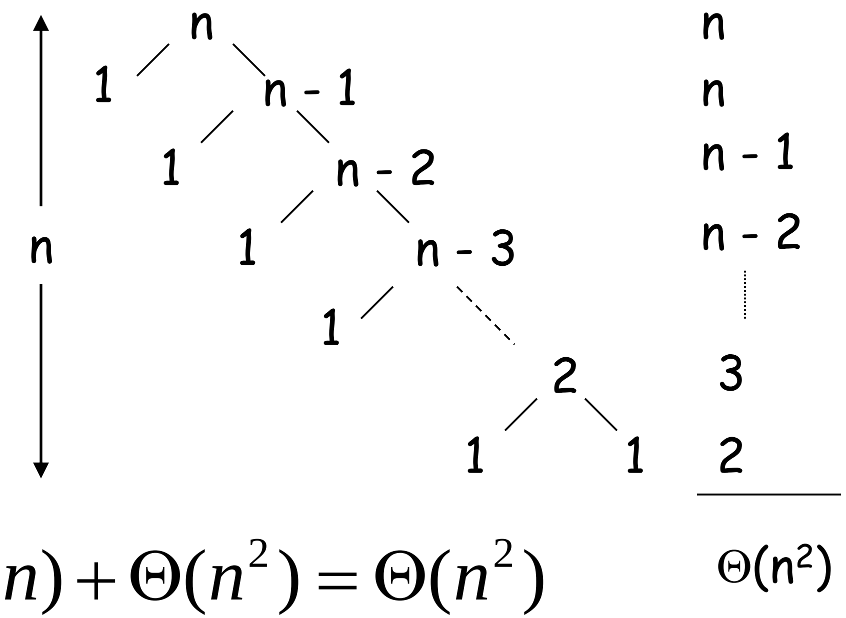
# Worst Case Partitioning

- **Worst-case partitioning**
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- **Recurrence: q=1**

T(n) = T(1) + T(n – 1) + n,

T(1) = Θ(1)

T(n) = T(n – 1) + n

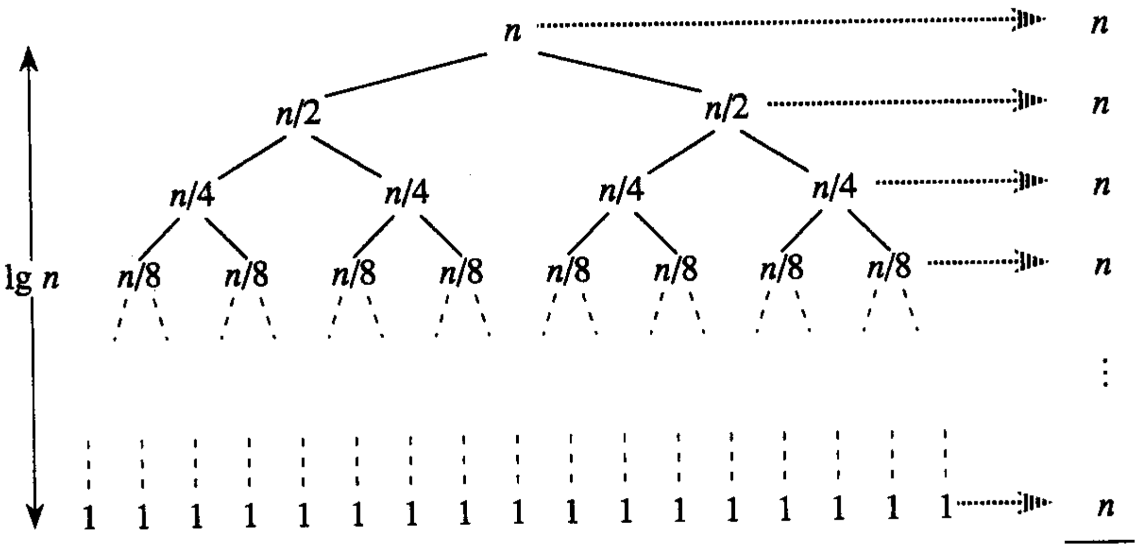



$$= n + \left( \sum_{k=1}^{n} k \right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

When does the worst case happen?

# Best Case Partitioning

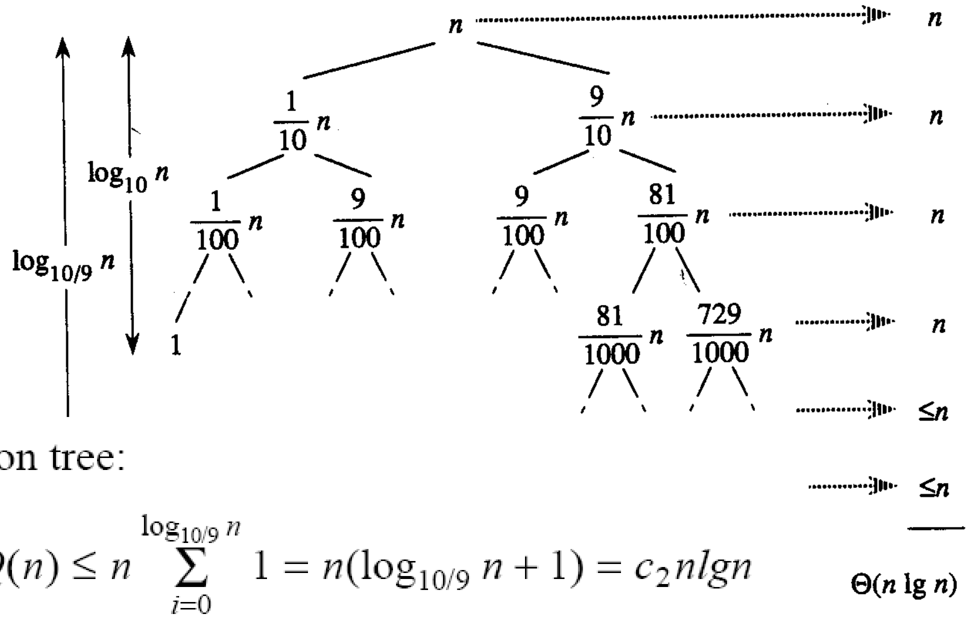
- **Best-case partitioning**
  - Partitioning produces two regions of size n/2



# Case Between Worst and Best

- 9-to-1 proportional split

$$Q(n) = Q(9n/10) + Q(n/10) + n$$




- Using the recursion tree:

longest path: $Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 n lgn$

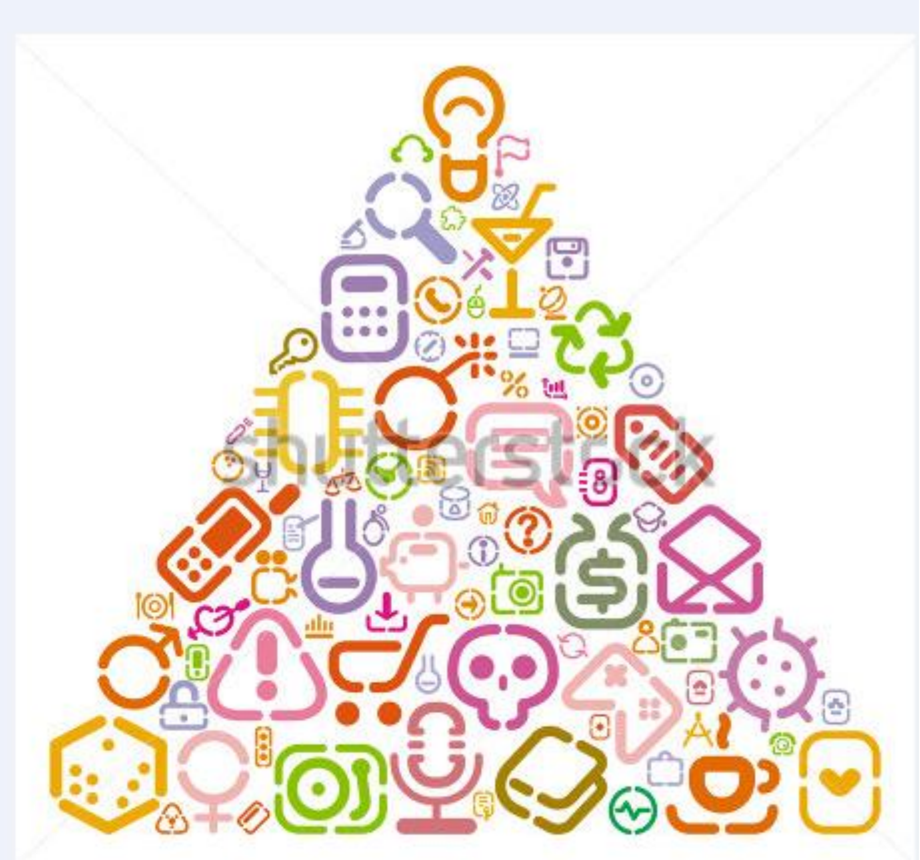
shortest path: $Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 n lgn$

Thus, $Q(n) = \Theta(nlgn)$

# Heap

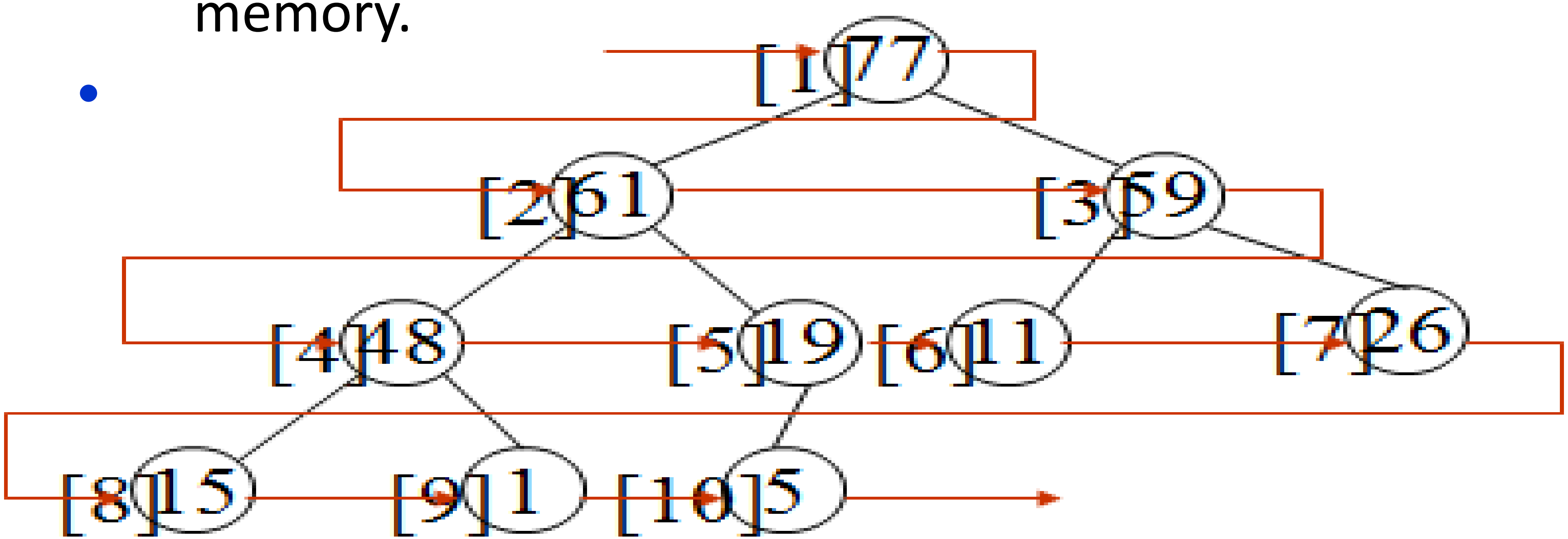
**Module I**

**Kiran Waghmare**



# Definition in Data Structure

- **Heap:**
  - A special form of complete binary tree that key value of each node is no smaller (larger) than the key value of its children (if any).
- **Max-Heap:**
  - root node has the largest key. A max tree is a tree in which the key value in each node is no smaller than the key values in its children. A max heap is a complete binary tree that is also a max tree.
- **Min-Heap:**
  - root node has the smallest key. A min tree is a tree in which the key value in each node is no larger than the key values in its children. A min heap is a complete binary tree that is also a min tree.
- **Complete Binary Tree:**
  - A complete binary tree is a binary tree in which every level, *except possibly the last*, **is completely filled, and all nodes are as far left as possible**

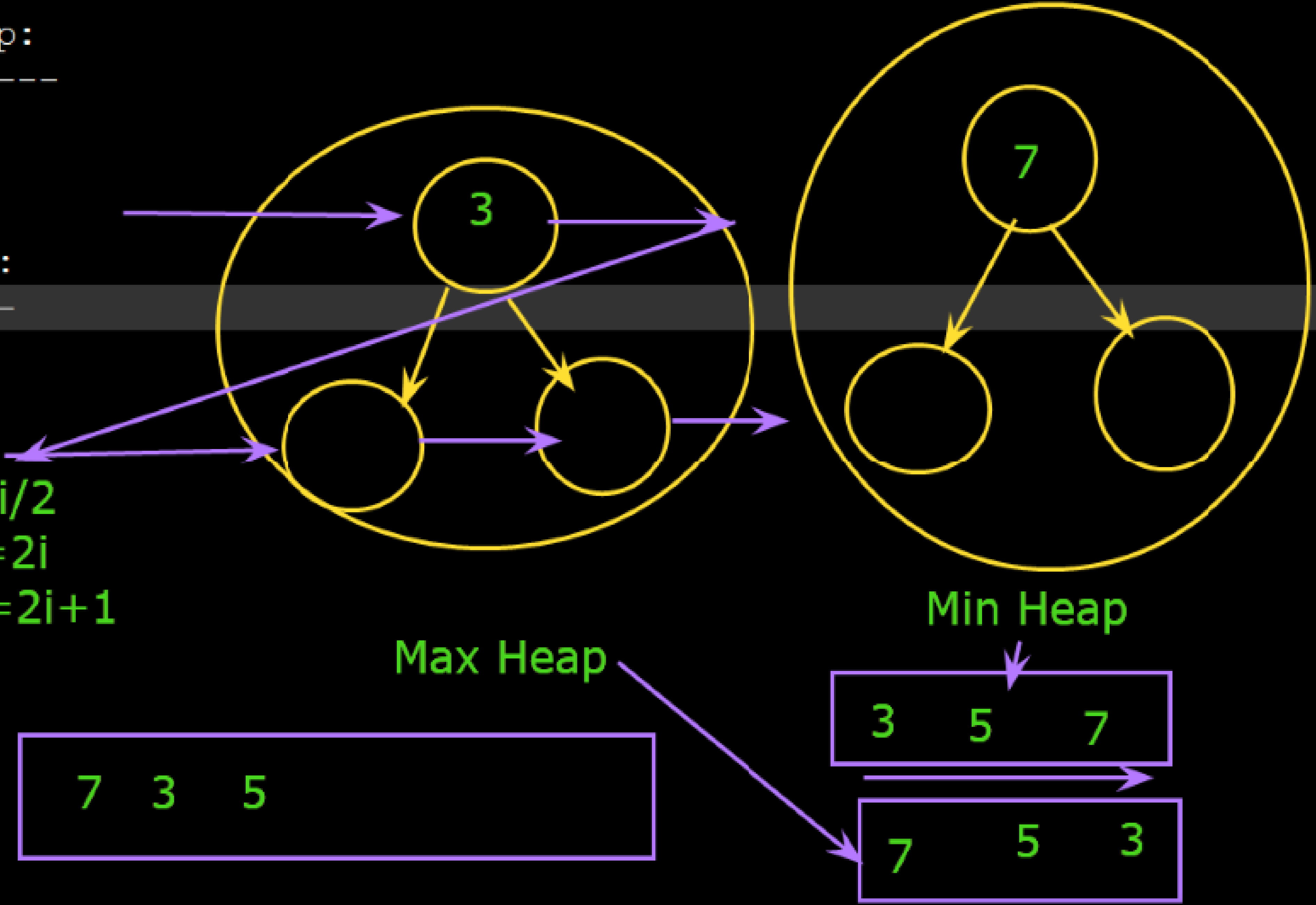
- **Note:**
  - Heap in data structure is a complete binary tree!
    - **(Nice representation in Array)**
  - Heap in C program environment is an array of memory.
- 


  – Stored using array in C

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 77 | 61 | 59 | 48 | 19 | 11 | 26 | 15 | 1 | 5 |

Types of Heap:
----------------
1. Max-Heap
2. Min-Heap

Heap Sorting:
--------------

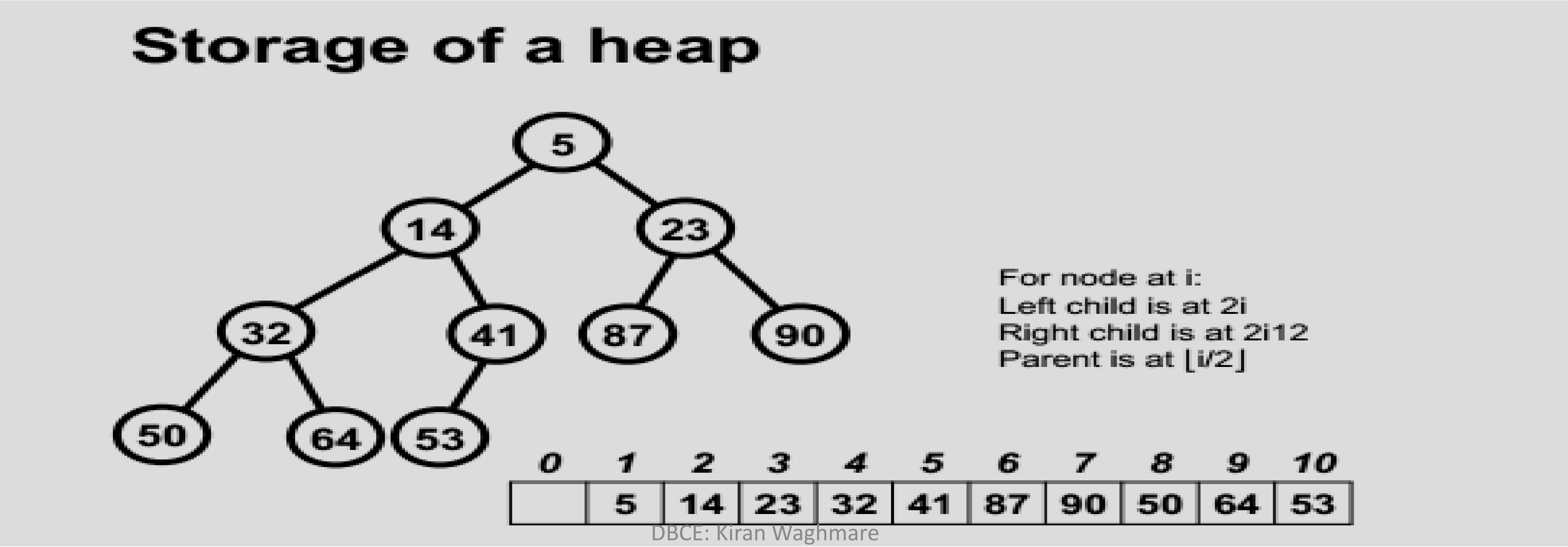



$P(node) = i/2$
$LC(node) = 2i$
$RC(node) = 2i+1$

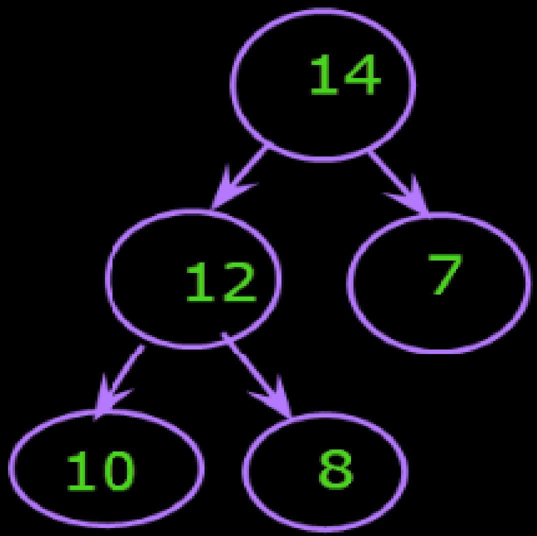
# Heap

**For any node n in position i :**

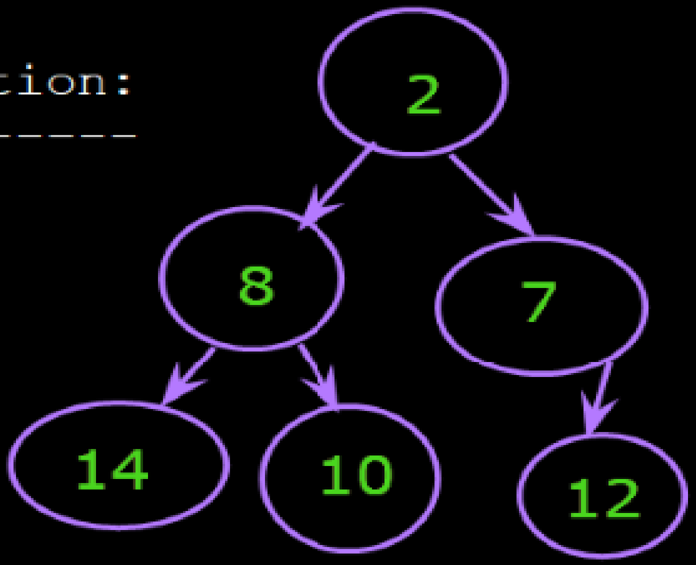
1. **LeftChild(i) : return 2i**
2. **RightChild(i) : return 2i+1**
3. **Parent(i) : return i/2**



```
Max-Heap:
Insertion operation:
----------------------
```




```
Min-Heap:
Insertion operation:
----------------------
```




```
Heap Sorting:
--------------
```

# Heap

- **Operations**
  - Creation of an empty heap
  - **Insertion** of a new element into the heap
  - **Deletion** of the largest(smallest) element from the heap

  - Heap is **complete binary tree**, can be represented by **array.**
- **So the complexity of**
  - inserting any node or
  - deleting the root node

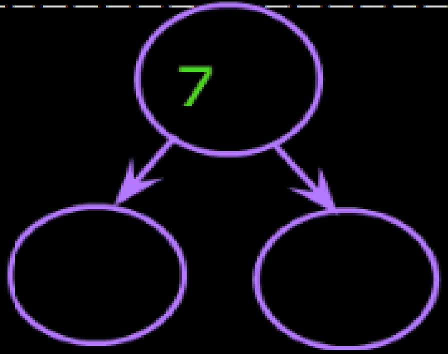
  from Heap is

$$\text{O(height)} = \text{O}(\log_2 n).$$

Max-Heap:
Insertion operation:
Deletion :Root node
--------------------

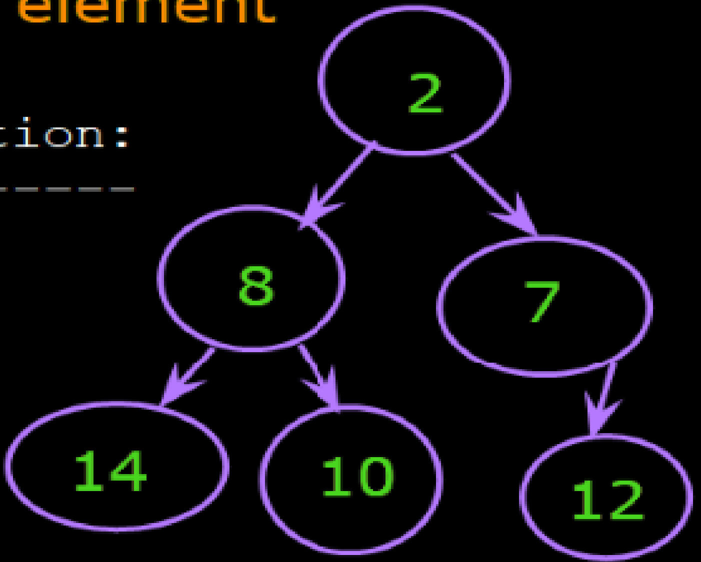
14 12 10 8 7 2

Desending order of elementsin max heap




last element

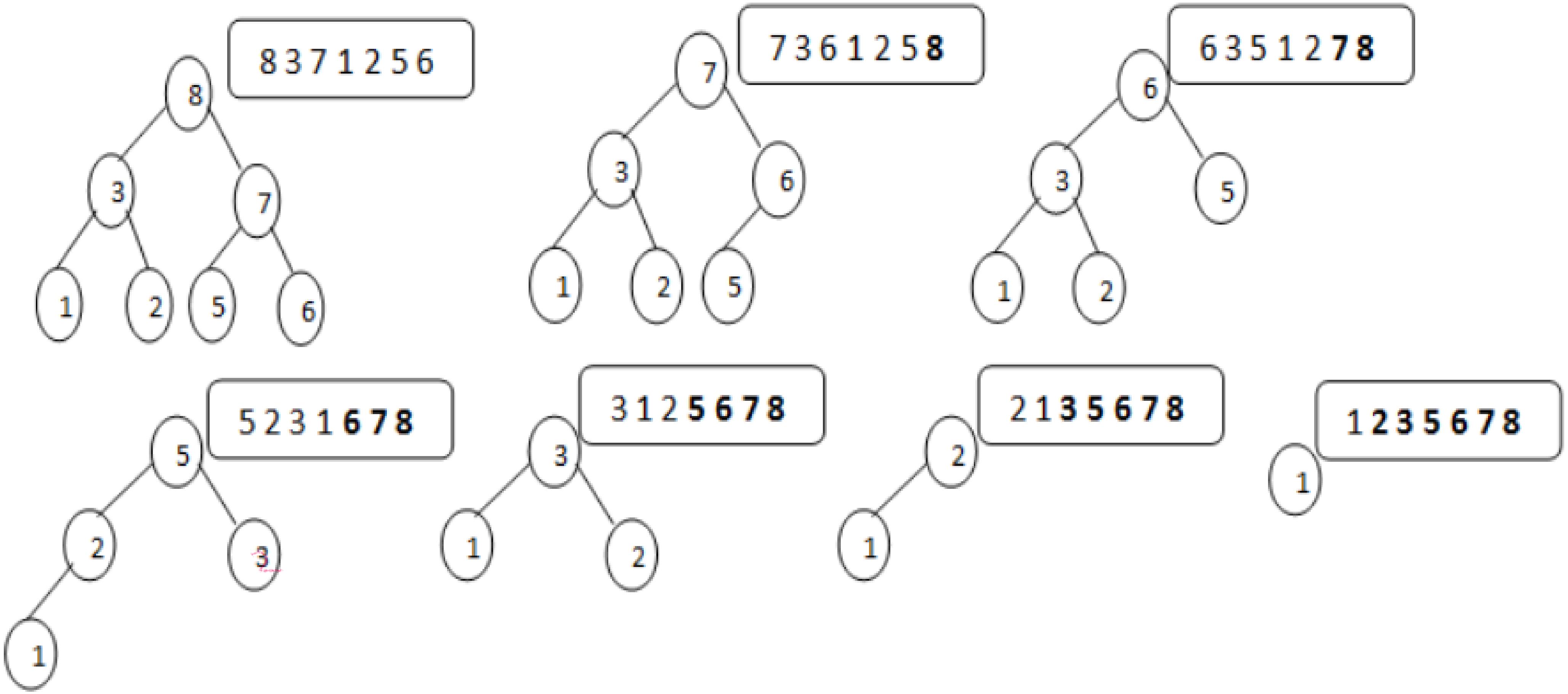
Min-Heap:
Insertion operation:
--------------------



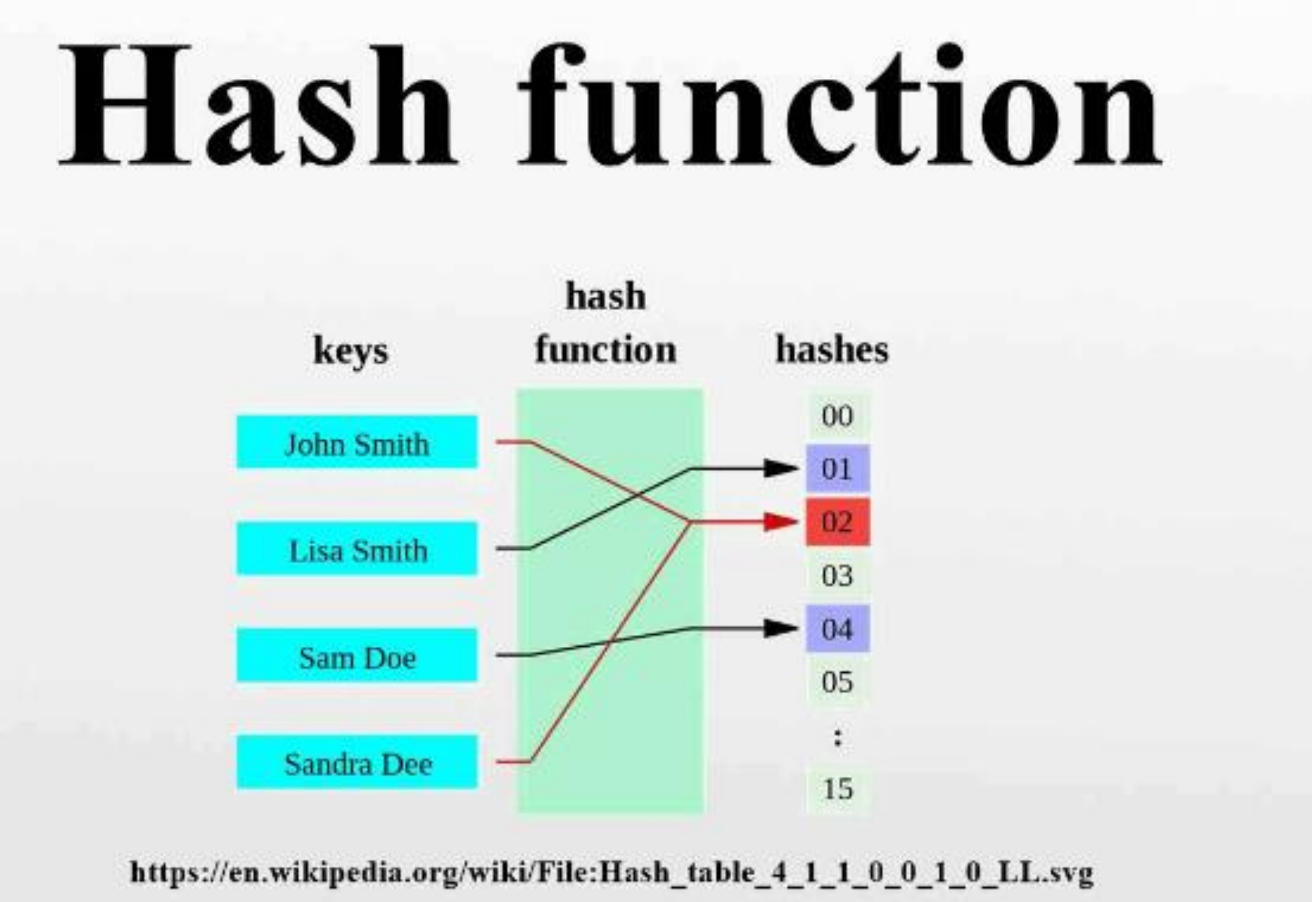

Heap Sorting:
-------------

## Example of Heap Sort:

**Example:-** The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



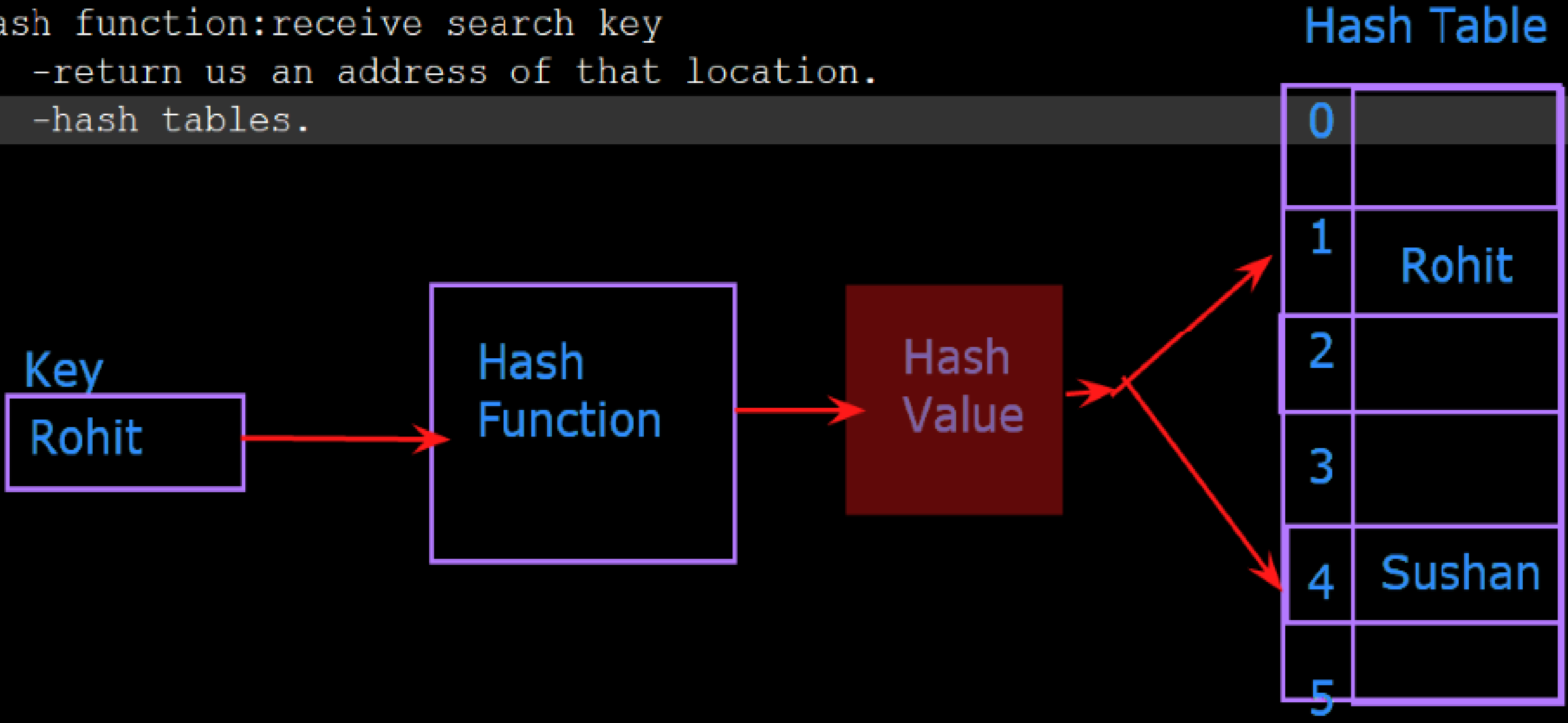
# Hashing

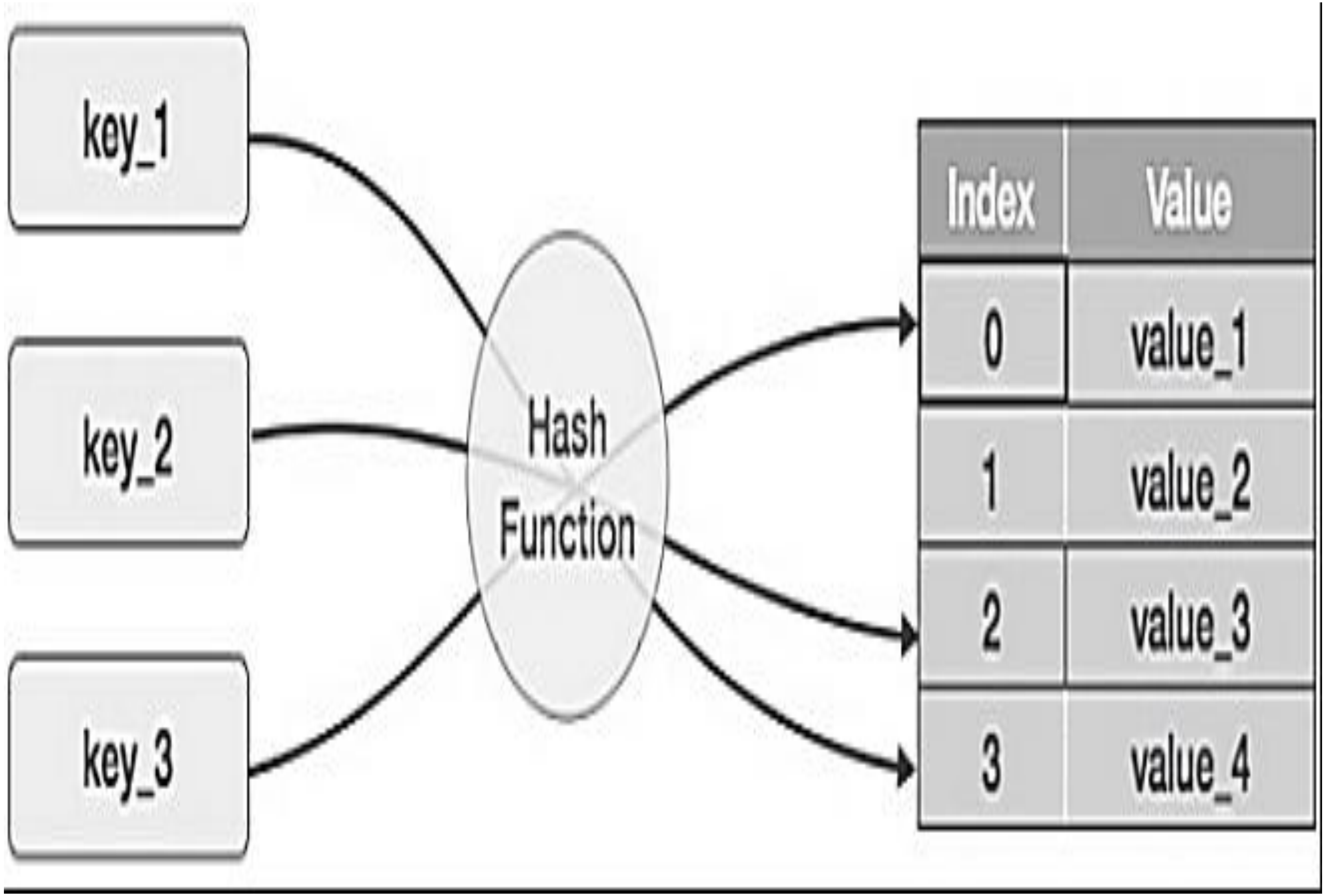



Unit 4

Kiran Waghmare

Hashing:
----------
-fastest Searching techniques.
-A techique tha tdetermines an index(location) of that data.
-hash function:receive search key
    -return us an address of that location.
    -hash tables.

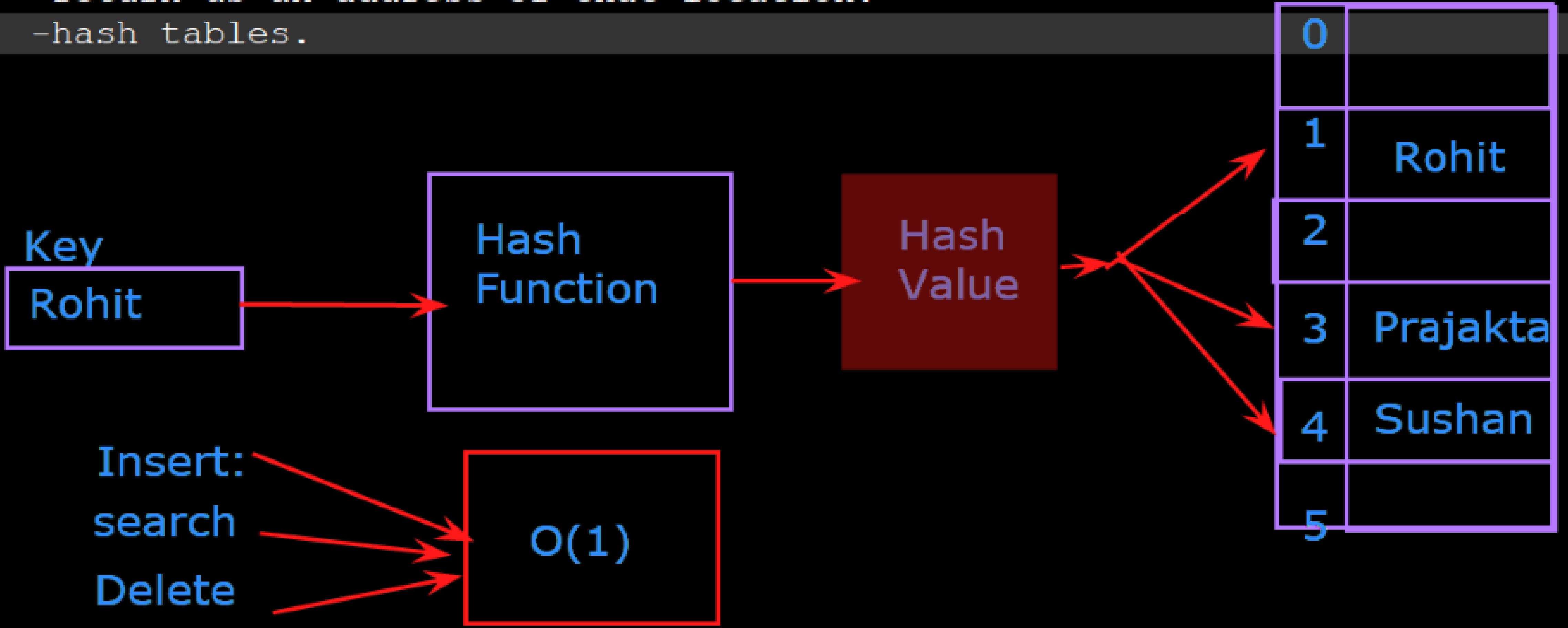


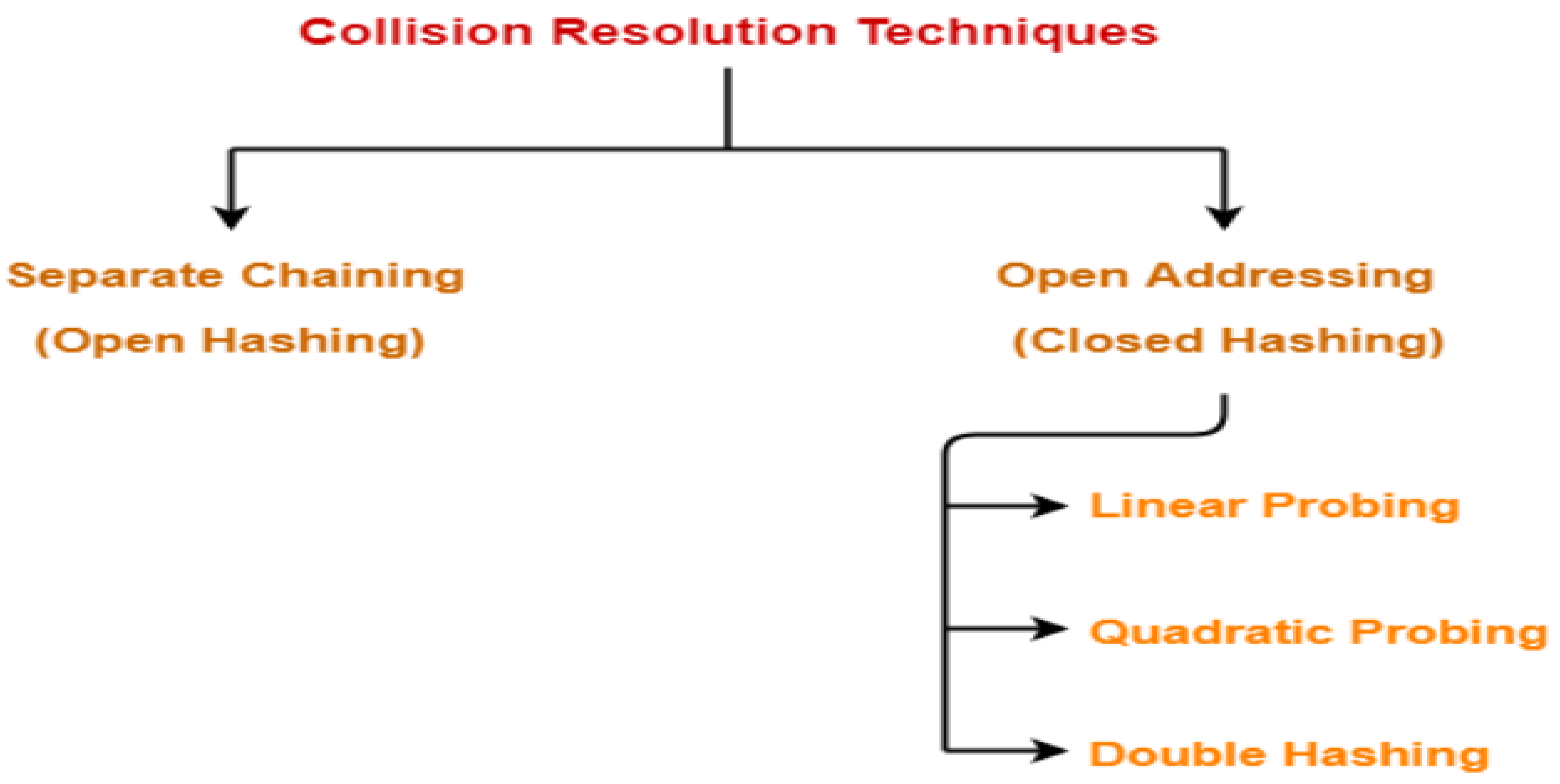

key_1
key_2
key_3
Hash
Function
Index
Value
0
value_1
1
value_2
2
value_3
3
value_4

# Hash Table

- A **hash table** is a data structure that stores elements and allows insertions, lookups, and deletions to be performed in O(1) time.
- A **hash table** is an alternative method for representing a dictionary
- In a hash table, a **hash function** is used to map keys into positions in a table. This act is called **hashing**
- Hash Table Operations
  - **Search**: compute f(k) and see if a pair exists
  - **Insert**: compute f(k) and place it in that position
  - **Delete**: compute f(k) and delete the pair in that position
- In ideal situation, hash table search, insert or delete takes $\Theta(1)$

```
Hashing:
---------
-fastest Searching techniques.
-A techique tha tdetermines an index(location) of that data.
-hash function:receive search key
    -return us an address of that location.
    -hash tables.
```



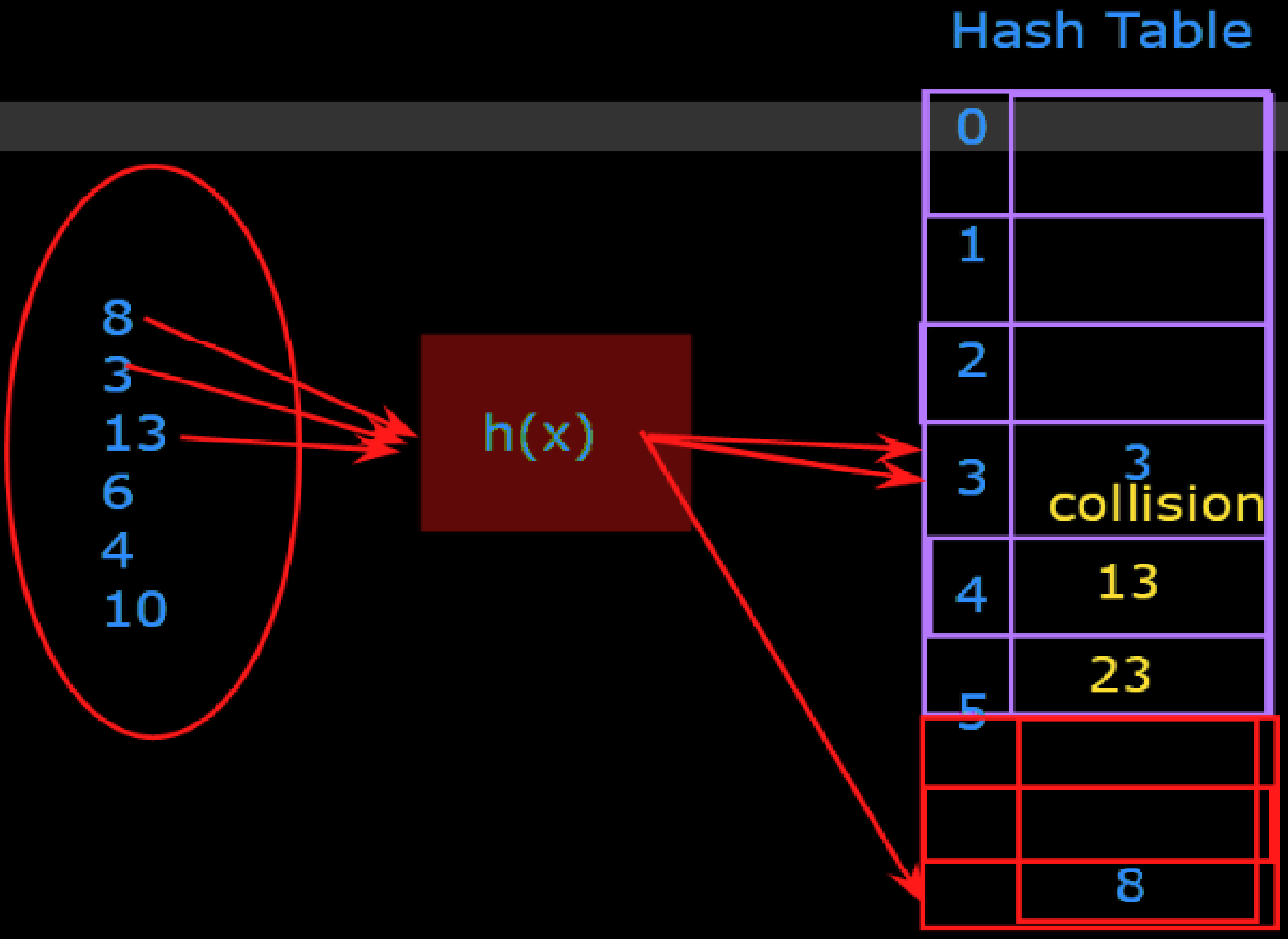

Collision Resolution Techniques
Separate Chaining
(Open Hashing)
Open Addressing
(Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

```
Types of Hashing:
---------------------
1. Open Hashing
    -Chaining ---> Linked list
2. Closed Hashing
    -Open Addressing:
        -Linear probing
        -Quadratic probing
        -Double Hashing

Hash function:

h(x)= x % i^2= x%20=
h(8)= 8%10 = 8
```

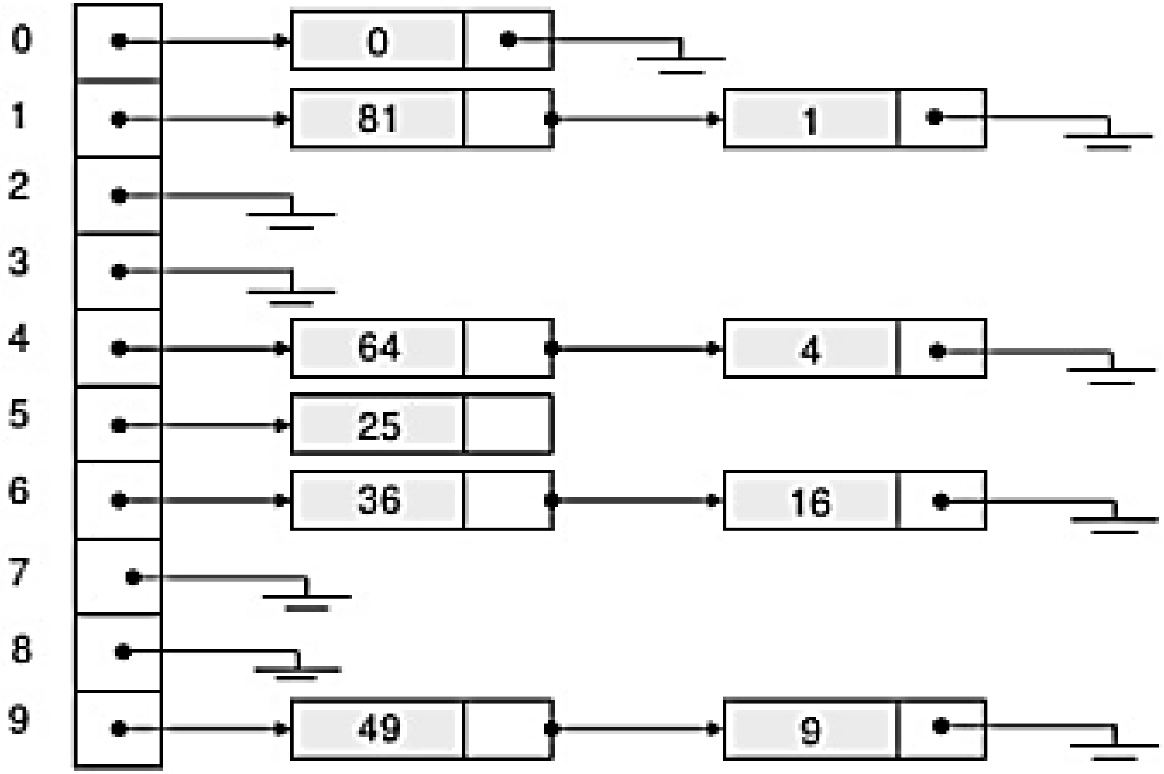

Sagar=> int



# Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.




Exercise: Represent the keys {89, 18, 49, 58, 69, 78} in hash table using separate chaining.

# Linear probing: example

## $h(k,n) = k \% n$

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

| | *After insert 89* | *After insert 18* | *After insert 49* | *After insert 58* | *After insert 9* |
|---|---|---|---|---|---|
| 0 | | | 49 | 49 | 49 |
| 1 | | | | 58 | 58 |
| 2 | | | | | 9 |
| 3 | | | | | |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

# Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **$h(x) = x \bmod 11$**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 → $2+1^2=3$
    - 25 mod 11 = 3 → $3+1^2=4$
    - 24 mod 11 = 2 → $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 → $9+1^2$, $9+2^2$ mod 11,
      $9+3^2$ mod 11 =7

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 2 |
| 3 | 13 |
| 4 | 25 |
| 5 | |
| 6 | 24 |
| 7 | 9 |
| 8 | 30 |
| 9 | 20 |
| 10 | 10 |

# Thanks