UNIT- IV	Searching And Sorting On Various Data Structures	(9Hrs)
Sequential Search - Binary Search - Comparison Trees - Breadth First Search - Depth First Search Insertion Sort - Selection Sort - Shell Sort - Divide and Conquer Sort - Merge Sort - Quick Sort- Heapsort - Introduction to Hashing		

Searching Algorithms

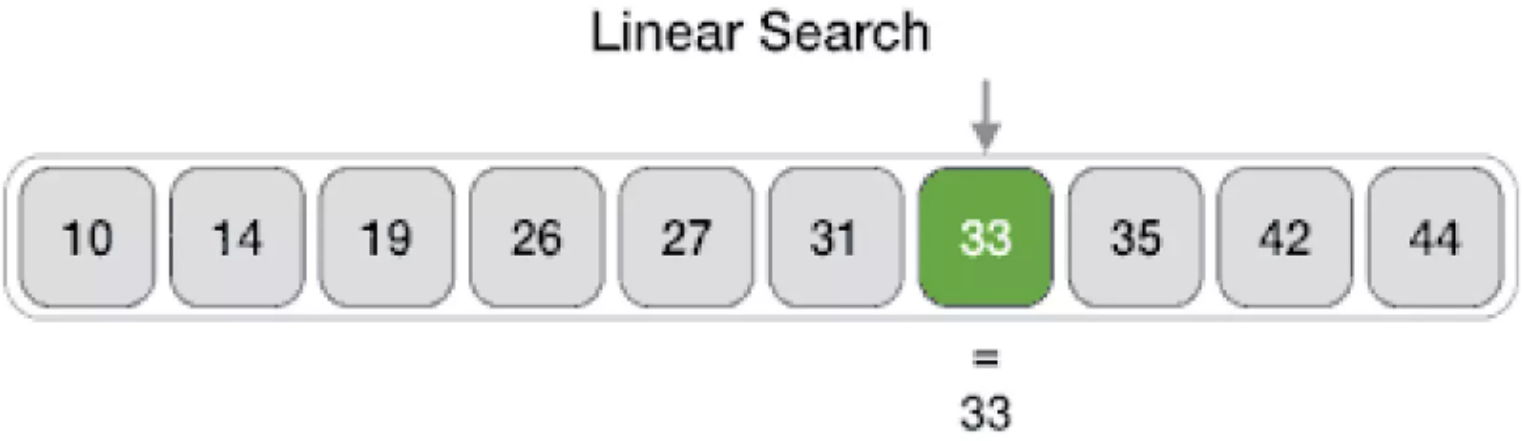
- Consider a database of banking system where information of all customers is stored, such as name, address and account number etc. If a manager wants to search for a record of a particular customer, he has to look for that record from among all records that has been stored in the database.
- This process of looking up for a particular record in a database is referred as **searching.**
- Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear search
2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the centre of the search structure and divide the search space in half. For Example: Binary search

Sequential/Linear Search

- Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- A simple approach is to do a **linear search**, i.e

1. Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
2. If x matches with an element, return the index.
3. If x doesn't match with any of elements, return -1.



Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)
    for each item in the list
       if match item == value
          return the item's location
       end if
    end for
end procedure
```

- **Problem:** Given an array arr[] of n elements, write a function to search a given element x in arr[].
- **Examples :**

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
          x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
           x = 175;
Output : -1
Element x is not present in arr[].
```

The **time complexity** of the above algorithm is O(n).

Binary Search

- The sequential search algorithm is very slow.
- If we have an array of 1000 elements, we must make 1000 comparisons in the worst case.
- The binary search starts by testing the data in the element at the middle of the array to determine if the target is in the first or the second half of the list.
- mid = (begin + end) / 2
- If it is in the first half, we do not need to check the second half.
- If it is in the second half, we do not need to test the first half.
- In other words, we eliminate half the list from further consideration with just one comparison. We repeat this process, eliminating half of the remaining list with each test, until we find the target or determine that it is not in the list.
- To find the middle of the list, we need three variables: one to identify the beginning of the list, one to identify the middle of the list, and one to identify the end of the list.
- We analyze two cases here: the target is in the list and the target is not in the list.

Binary Search

- **Binary Search:** Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
- We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half sub array after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

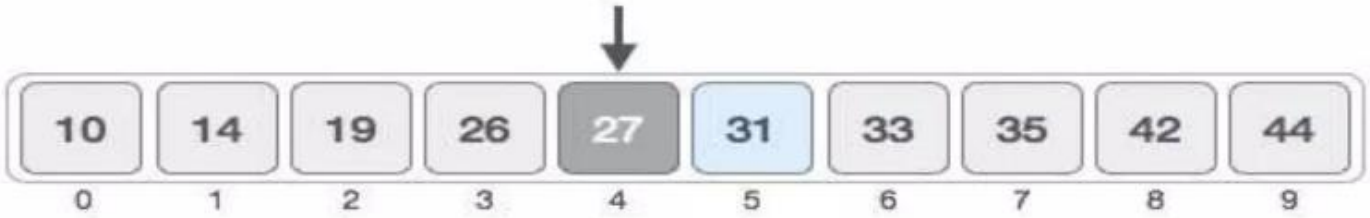
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

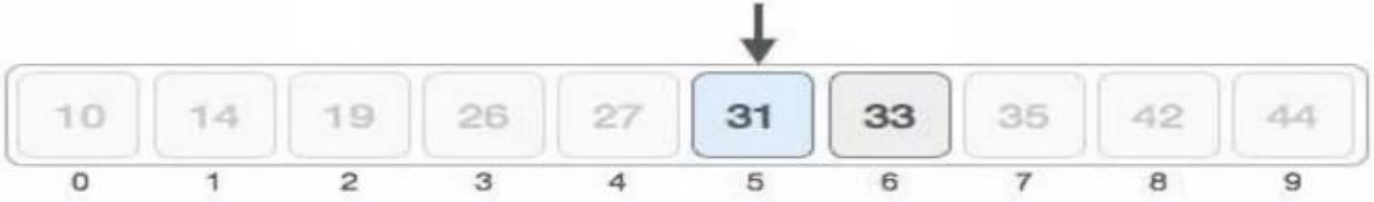
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

Comparison Trees

Differences between Binary tree and Binary search tree

Basis for comparison	Binary tree	Binary search tree
Definition	A binary tree is a non-linear data structure in which a node can have utmost two children, i.e., a node can have 0, 1 or maximum two children.	A binary search tree is an ordered binary tree in which some order is followed to organize the nodes in a tree.
Structure	The structure of the binary tree is that the first node or the topmost node is known as the root node. Each node in a binary tree contains the left pointer and the right pointer. The left pointer contains the address of the left subtree, whereas right pointer contains the address of right subtree.	The binary search tree is one of the types of binary tree that has the value of all the nodes in the left subtree lesser or equal to the root node, and the value of all the nodes in a right subtree are greater than or equal to the value of the root node.
Operations	The operations that can be implemented on a binary tree are insertion, deletion, and traversal.	Binary search trees are the sorted binary trees that provide fast insertion, deletion and search. Lookups mainly implement binary search as all the keys are arranged in sorted order.
types	Four types of binary trees are Full Binary Tree, Complete Binary Tree, Perfect Binary Tree, and Extended Binary Tree.	There are different types of binary search trees such as AVL trees, Splay tree, Tango trees, etc.

Differences between Binary Search tree and AVL tree

Binary Search tree	AVL tree
Every binary search tree is a binary tree because both the trees contain the utmost two children.	Every AVL tree is also a binary tree because AVL tree also has the utmost two children.
In BST, there is no term exists, such as balance factor.	In the AVL tree, each node contains a balance factor, and the value of the balance factor must be either -1, 0, or 1.
Every Binary Search tree is not an AVL tree because BST could be either a balanced or an unbalanced tree.	Every AVL tree is a binary search tree because the AVL tree follows the property of the BST.
Each node in the Binary Search tree consists of three fields, i.e., left subtree, node value, and the right subtree.	Each node in the AVL tree consists of four fields, i.e., left subtree, node value, right subtree, and the balance factor.
In the case of Binary Search tree, if we want to insert any node in the tree then we compare the node value with the root value; if the value of node is greater than the root node value then the node is inserted to the right subtree otherwise the node is inserted to the left subtree. Once the node is inserted, there is no need of checking the height balance factor for the insertion to be completed.	In the case of AVL tree, first, we will find the suitable place to insert the node. Once the node is inserted, we will calculate the balance factor of each node. If the balance factor of each node is satisfied, the insertion is completed. If the balance factor is greater than 1, then we need to perform some rotations to balance the tree.
Searching is inefficient in BST when there are large number of nodes available in the tree because the height is not balanced.	Searching is efficient in AVL tree even when there are large number of nodes in the tree because the height is balanced.

Differences between Binary Search tree and AVL tree

Binary Search tree	AVL tree
In Binary Search tree, the height or depth of the tree is O(n) where n is the number of nodes in the Binary Search tree.	In AVL tree, the height or depth of the tree is O(logn).
It is simple to implement as we have to follow the Binary Search properties to insert the node.	It is complex to implement because in AVL tree, we have to first construct the AVL tree, and then we need to check height balance. If the height is imbalance then we need to perform some rotations to balance the tree.
BST is not a balanced tree because it does not follow the concept of the balance factor.	AVL tree is a height balanced tree because it follows the concept of the balance factor.
Searching is inefficient in BST when there are large number of nodes available in the tree because the height is not balanced.	Searching is efficient in AVL tree even when there are large number of nodes in the tree because the height is balanced.

Differences between B tree and B+ tree

B tree	B+ tree
In the B tree, all the keys and records are stored in both internal as well as leaf nodes.	In the B+ tree, keys are the indexes stored in the internal nodes and records are stored in the leaf nodes.
In B tree, keys cannot be repeatedly stored, which means that there is no duplication of keys or records.	In the B+ tree, there can be redundancy in the occurrence of the keys. In this case, the records are stored in the leaf nodes, whereas the keys are stored in the internal nodes, so redundant keys can be present in the internal nodes.
In the Btree, leaf nodes are not linked to each other.	In B+ tree, the leaf nodes are linked to each other to provide the sequential access.
In Btree, searching is not very efficient because the records are either stored in leaf or internal nodes.	In B+ tree, searching is very efficient or quicker because all the records are stored in the leaf nodes.
Deletion of internal nodes is very slow and a time-consuming process as we need to consider the child of the deleted key also.	Deletion in B+ tree is very fast because all the records are stored in the leaf nodes so we do not have to consider the child of the node.
In Btree, sequential access is not possible.	In the B+ tree, all the leaf nodes are connected to each other through a pointer, so sequential access is possible.
In Btree, the more number of splitting operations are performed due to which height increases compared to width,	B+ tree has more width as compared to height.
In Btree, each node has atleast two branches and each node contains some records, so we do not need to traverse till the leaf nodes to get the data.	In B+ tree, internal nodes contain only pointers and leaf nodes contain records. All the leaf nodes are at the same level, so we need to traverse till the leaf nodes to get the data.
The root node contains atleast 2 to m children where m is the order of the tree.	The root node contains atleast 2 to m children where m is the order of the tree.

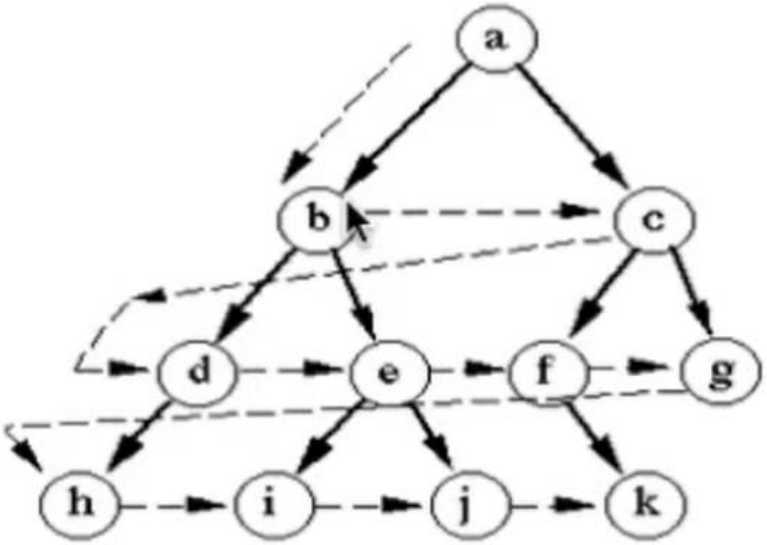
Breadth First Search

Graph Traversal (Search)

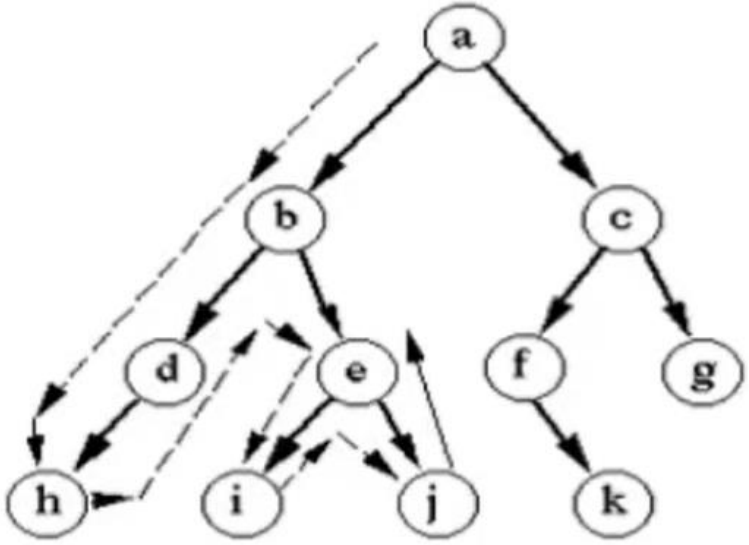
- ✓ Finds **the edges** to be used in the **search process without** creating **loops.**
- ✓ The process of visiting (checking and/or updating) each vertex in a **graph exactly once.**
- ✓ **traversals** are classified by **the order** in which the vertices are **visited.**

Graph Traversal (Search)

- ✓Two different types of traversal strategies are:
  - ✓Breadth First Search (BFS)
  - ✓Depth First Search (DFS)



Breadth-first search



Depth-first search

Breadth First Search (BFS)

✓**Visit the sibling before the child.**

➢Use **Queue** data structure

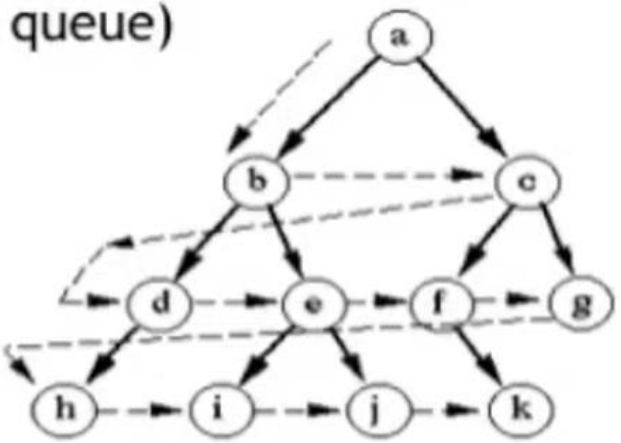
Steps:

Do until the queue becomes empty & all the nodes are visited.

- **Enqueue** the node into queue

➢Make the front element visited, Enqueue its neighbors into the queue without duplicates (either visited or already in queue)

➢**Dequeue** the front element.



Breadth-first search

BFS

- Step1 : choose any node in the graph, designate it as the search node & mark it as visited
- Setp 2: Using the adjacency matrix of the graph, find all the univisted adjacent node and enqueue them into queue Q
- Step 3: then the node is dequeued from the queue. Mark that node as visited and designate it as the new search node
- Step 4: Repeat the step 2 and 3 using the new search node
- Step 5: this process continues until the queue Q which keeps track of the adjacent nodes is empty.

Example:

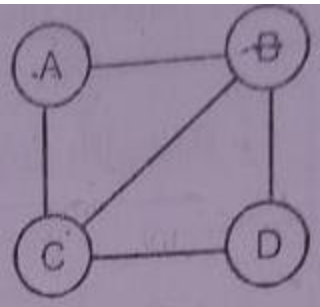


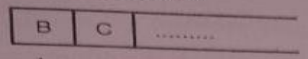
Figure 7.4.1

Adjacency matrix

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

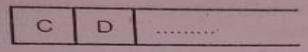
Figure 7.4.2

1. Let 'A' be the source vertex. Mark it to as visited.
2. Find the adjacent unvisited vertices of 'A' and enqueue then into the queue. Here B and C are adjacent nodes of A



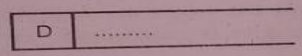
and B and C are enqueued.

3. Then vertex 'B' is dequeued and its adjacent vertices C and D are taken from the adjacency matrix for enqueuing. Since vertex C is already in the queue, vertex D alone is enqueued.



Here B is dequeued, D is enqueued.

4. Then vertex 'C' is dequeued and its adjacent vertices A, B and D are found out. Since vertices A and B are already visited and vertex D is also in the queue, no enqueue operation takes place.



Here C is dequeued

5. Then vertex 'D' is dequeued. This process terminates as all the vertices are visited and the queue is also empty.

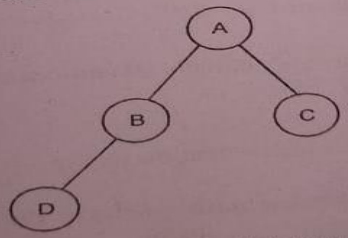


Figure 7,4.3: Breadth first spanning tree

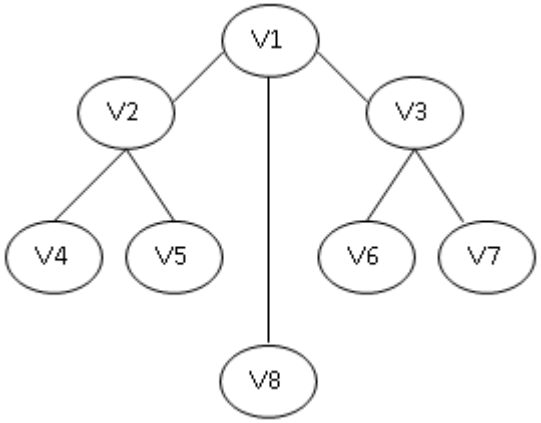
Applications of breadth first search

1. To check whether the graph is connected or not.

Algorithm BFS gives the details.

Procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked as VISITED(I) = 1. The graph G and array VISITED are global and VISITED is initialised to 0.//
VISITED(v) ← 1
Initialise Q to be empty //Q is a queue//
loop
for all vertices w adjacent to v do
if VISITED(w) = 0 //add w to queue//
then [call ADDQ(w, Q); VISITED(w) ← 1] //mark w as VISITED//
end
if Q is empty then return
call DELETEQ(v,Q)
forever
end BFS



Let us consider the same example, given in figure. We start say, with V_1. Its adjacent vertices are V_2, V_8 , V_3 . we visit all one by one. We pick on one of these, say V_2. The unvisited adjacent vertices to V_2 are V_4, V_5 . we visit both . we back to the remaining visited vertices of V_1 and pick on one of this, say V_3. T The unvisited adjacent vertices to V_3 are V_6,V_7. There are no more unvisited adjacent vertices of V_8, V_4, V_5, V_6 and V_7. Thus the sequence so generated is $V_{1,}V_{2,}$ $V_{8,}$ $V_{3,}V_4$, $V_{5,}V_{6,}$ V_7.

Applications of Breadth First Traversal

- ✓ **Shortest Path and Minimum Spanning Tree** for unweighted graph
- ✓ Ford-Fulkerson algorithm
- ✓ In Garbage Collection - **Cheney's algorithm.**
- ✓ to find all **neighbor nodes in Peer to Peer** Networks (like BitTorrent,
- ✓ **Crawlers** in Search Engines
- ✓ **Social Networking Websites** - find people within a given distance 'k'
- ✓ **GPS** Navigation systems:
- ✓ **Broadcasting** in Network
- ✓ **Cycle detection** in undirected graph
- ✓ To test if a graph is **Bipartite**
- ✓ Path Finding
- ✓ Finding all nodes within one connected component:

Depth First Search (DFS)

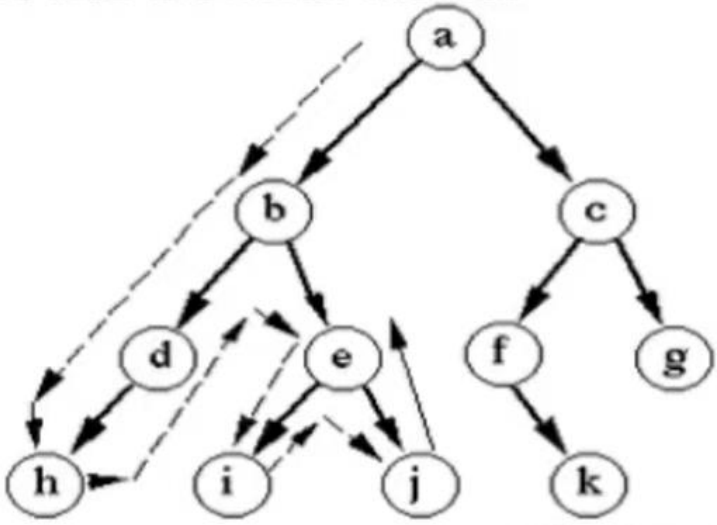
✓**Visit the Child before the sibling.**

➢Use **Stack** data structure

Steps:

Do until the queue becomes empty & all the nodes are visited:

- **Push** the node into stack
- ➢Make the top element visited,
- ➢**Pop** it, make it visited and then push its neighbors into the stack without duplicates (either visited or already in queue)



DEPTH FIRST SEARCH

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

This procedure is best described recursively as in

Procedure DFS(v)
// Given an undirected graph G = (V.E) with n vertices and an array visited (n) initially set to zero . This algorithm visits all vertices reachable from v .G and VISITED are global > //VISITED (v) ← 1

for each vertex w adjacent to v do
if VISITED (w) =0 then call DFS (w)

end

end DFS

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

To implement the Depthfirst Search perform the following Steps :

Step : 1 Choose any node in the graph. Designate it as the search node and mark it as visited.

Step : 2 Using the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.

Step : 3 Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.

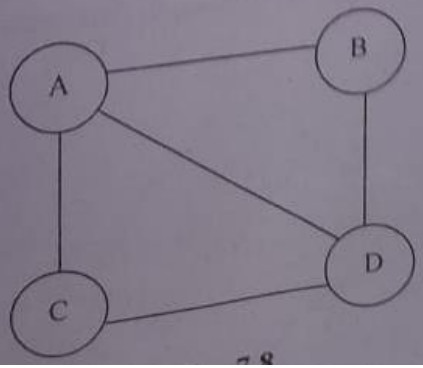
Step : 4 When a return to the previous search node in (3) is impossible, the search from the originally choosen search node is complete.

Step : 5 If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

ROUTINE FOR DEPTH FIRST SEARCH

```
Void DFS (Vertex V)
{
        visited [V] = True;
        for each W adjacent to V
                if (! visited [W])
                        DFS (W);
}
```

Example : -

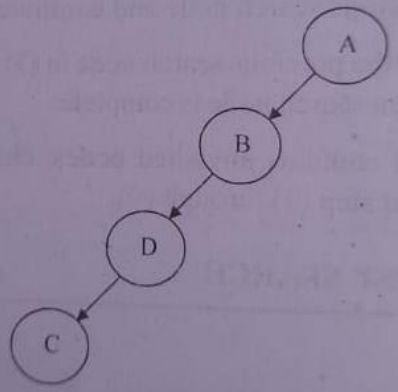


Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

Implementation

1. Let 'A' be the source vertex. Mark it to be visited.
2. Find the immediate adjacent unvisited vertex 'B' of 'A' Mark it to be visited.
3. From 'B' the next adjacent vertex is 'd' Mark it has visited.
4. From 'D' the next unvisited vertex is 'C' Mark it to be visited.

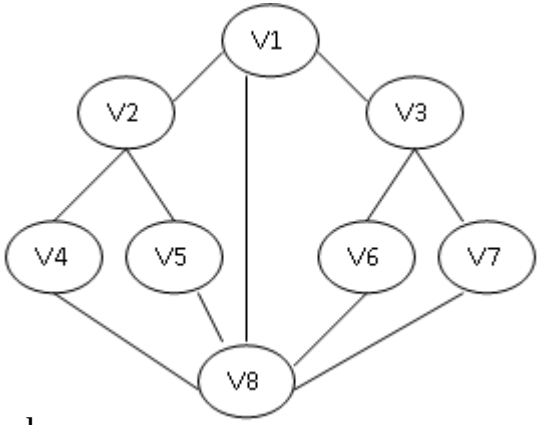


Depth First Spanning Tree

Applications of Depth First Search

1. To check whether the undirected graph is connected or not.
2. To check whether the connected undirected graph is Bioconnected or not.
3. To check the a Acyclicity of the directed graph.

Example



Let us start with $V_{1.}$

Its adjacent vertices are V2, V8, and V3. Let us pick on v2.

Its adjacent vertices are V1, V4, V5, V1 is already visited . let us pick on V4.

Its adjacent vertices are V2, V8.

V2 is already visited .let us visit V8.

Its adjacent vertices are V4, V5, V1, V6, V7.

V4 and V1 are visited. Let us traverse V5.

Its adjacent vertices are V2, V8. Both are already visited therefore, we back track.

We had V6 and V7 unvisited in the list of V8. We may visit any. We may visit any. We visit V6.

Its adjacent are V8 and V3. Obviously the choice is V3.

Its adjacent vertices are V1, V7 . We visit V7.

All the adjacent vertices of V7 are already visited, we back track and find that we have visited all the vertices.

Therefore the sequence of traversal is

V1, V2, V4, V5, V6, V3, V7.

This is not a unique or the only sequence possible using this traversal method.

We may implement the Depth First search by using a stack,pushing all unvisited vertices to the one just visited and poping the stack to find the next vertex to visit.

Applications of Depth First Traversal

- ✓ **Solving puzzles with only one solution (eg mazes)**
- ✓ For a weighted graph, DFS traversal of the graph produces the **minimum spanning tree and all pair shortest path tree.**
- ✓ **Detecting cycle** in a graph
- ✓ **Path Finding**
- ✓ **Topological Sorting**
- ✓ To test if a graph is **bipartite**
- ✓ Finding **Strongly Connected Components** of a graph

Breadth First Search (BFS) Vs Depth First Search (DFS)

BFS	DFS
Level based search	Depth Based Search
vertex based technique	edge based technique
Queue Data Structure is used	Stack Data Structure is used
BFS is more suitable for searching target vertices which are closer to the given source.	DFS is more suitable when there are solutions or targets away from source.
Not suitable for decision making trees (games, puzzles) since neighbors are considered	Suitable for decision making trees (games, puzzles)
Higher memory requirement than DFS	lower memory requirements because it's not necessary to store all of the child pointers at each level
BFS is slower	DFS is faster
Time complexity of BFS & DFS is O(V + E), where V is vertices & E is edges.	

Differences between BFS and DFS

	BFS	DFS
Full form	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.
Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.
Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Suitability for decision tree	It is not suitable for the decision tree because it requires exploring all the neighboring nodes first.	It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.

Insertion Sort

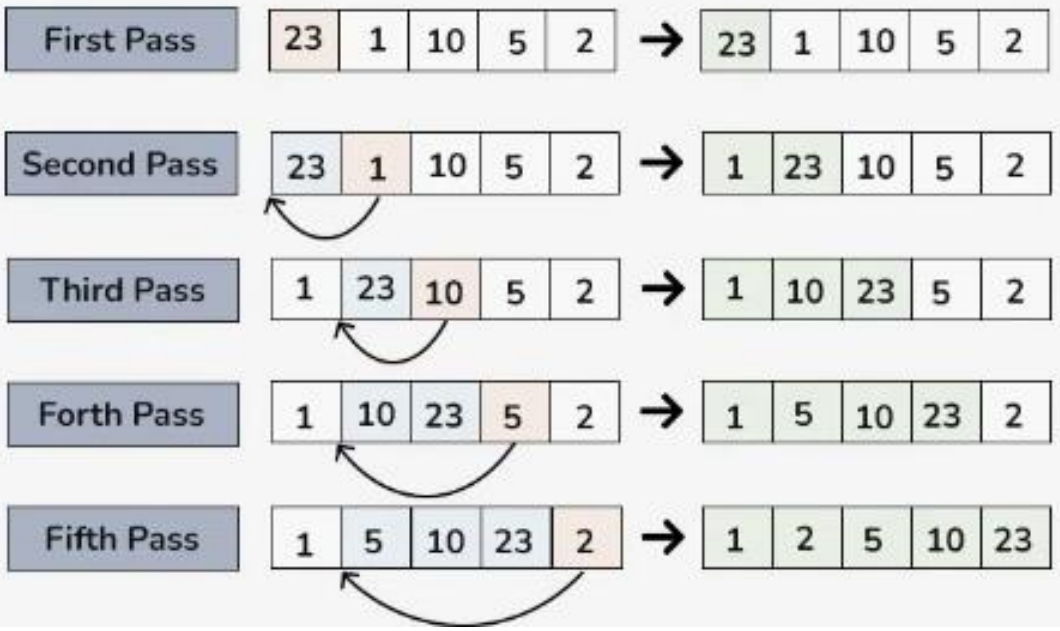
- **Insertion sort** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a **stable sorting** algorithm, meaning that elements with equal values maintain their relative order in the sorted output.
- **Insertion sort** is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

Insertion Sort

- During first iteration element in pos 1 is compared with element in pos 0 and insert it in the proper position
- During second iteration element in pos 2 is compared with elements in pos 1 and 0 and insert it in the proper position
- During third iteration element in pos 3 is compared with elements in pos 2, 1 and 0. Then insert it in the proper position
- In general in every iteration an element is compared with all elements before it and insert it in the proper position

Working of Insertion Sort Algorithm:

Consider an array having elements: ***{23, 1, 10, 5, 2}***



Insertion Sort

Algorithm InsertionSort(A, n)

1. for i=1 to n-1 do
    1. temp=A[i]
    2. j=i-1
    3. While j >= 0 and A[j] > temp do
        1. A[j+1] = A[j]
        2. j=j-1
    4. A[j+1] = temp

Selection Sort

- **Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Selection Sort

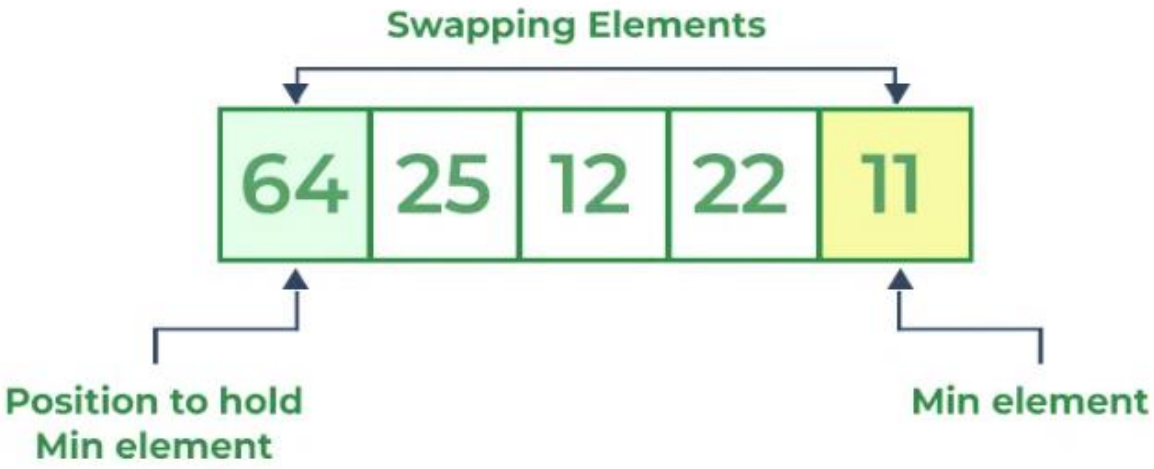
- In this method find the smallest element in the array and this element is stored in the first position. Then find the second smallest element and it is stored in the second position, and so on

How does Selection Sort Algorithm work?

Lets consider the following array as an example: ***arr[] = {64, 25, 12, 22, 11}***

First pass:

- *For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where* ***64*** *is stored presently, after traversing whole array it is clear that* ***11*** *is the lowest value.*
- *Thus, replace 64 with 11. After one iteration* ***11****, which happens to be the least value in the array, tends to appear in the first position of the sorted list.*

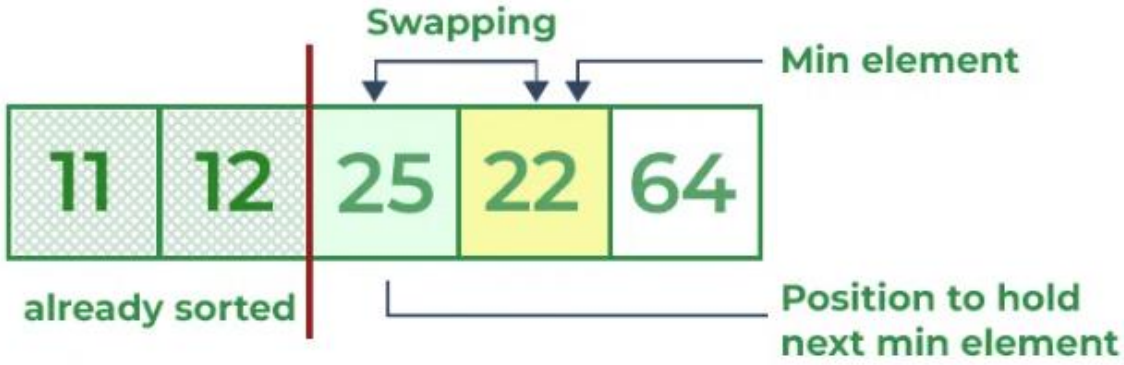


Second Pass:

- *For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.*
- *After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.*

Third Pass:

- *Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.*
- *While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.*



Fourth pass:

- *Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array*
- *As **25** is the 4th lowest value hence, it will place at the fourth position.*

Fifth Pass:

- *At last the largest value present in the array automatically get placed at the last position in the array*
- *The resulted array is the sorted array.*

Sorted array

Selection Sort

Algorithm SelectionSort(A, n)

```
{
    for i=0 to n-2 do
    {
        for j=i+1 to n-1 do
        {
                If A[i]>A[j] then
                        Swap A[i] and A[j]
        }
    }
}
```

Shell Sort

Introduction:

- Founded by Donald Shell and named the sorting algorithm after himself in 1959.
- 1st algorithm to break the quadratic time barrier but few years later, a sub quadratic time bound was proven.
- Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared.

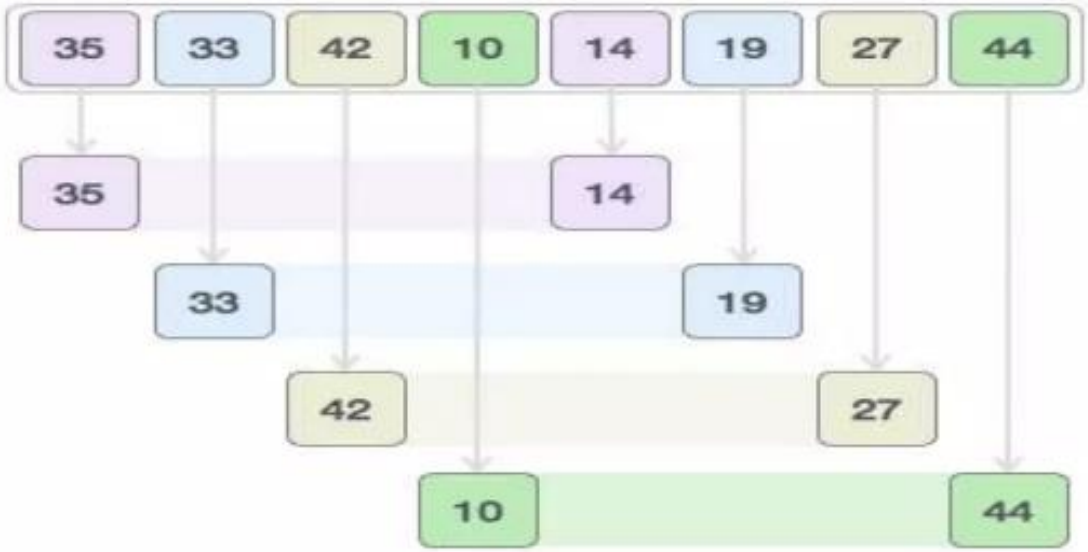
What is Shell Sort:

- **Shell sort** is a **sorting** algorithm. It is an in-place comparison **sort** and one of the oldest **sorting** algorithm. **Shell sort** is a generalization of insertion **sort** that allows the exchange of items that are far apart.
- Shell sort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.
- Shell sort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

Example:

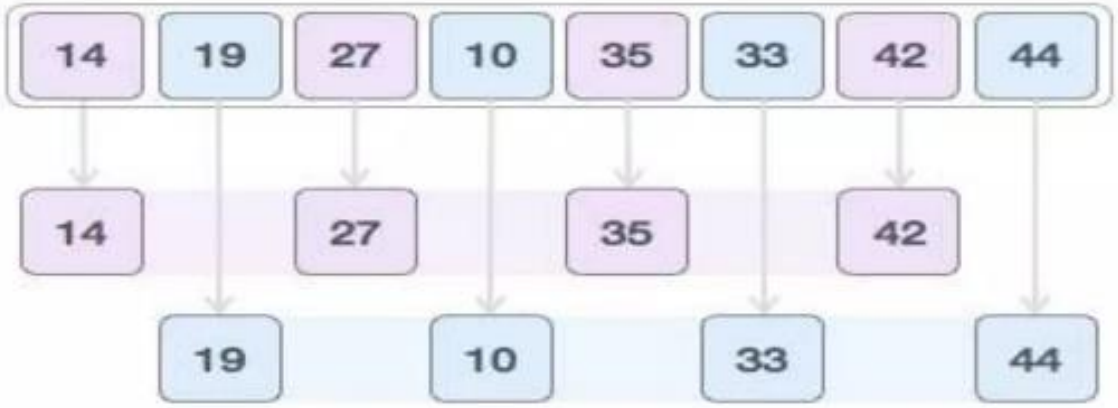
How Shell Sort Works?

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −

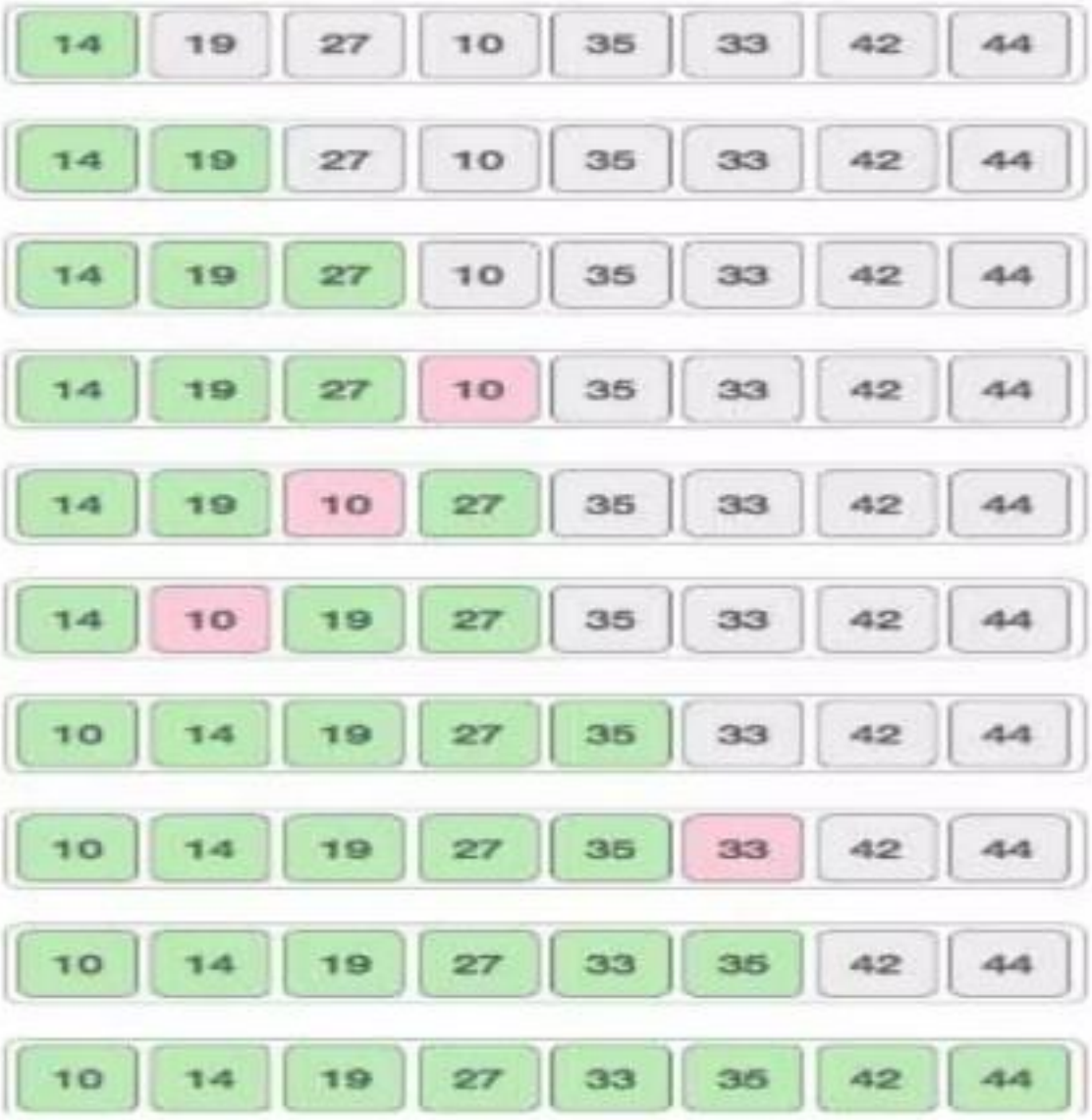
Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −

Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



Algorithm

The simple steps of achieving the shell sort are listed as follows -

```
ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
for (interval = n/2; interval > 0; interval /= 2)
for ( i = interval; i < n; i += 1)
temp = a[i];
for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
a[j] = a[j - interval];
a[j] = temp;
End ShellSort
```

Divide and Conquer Sort

Merge Sort

Definition:-

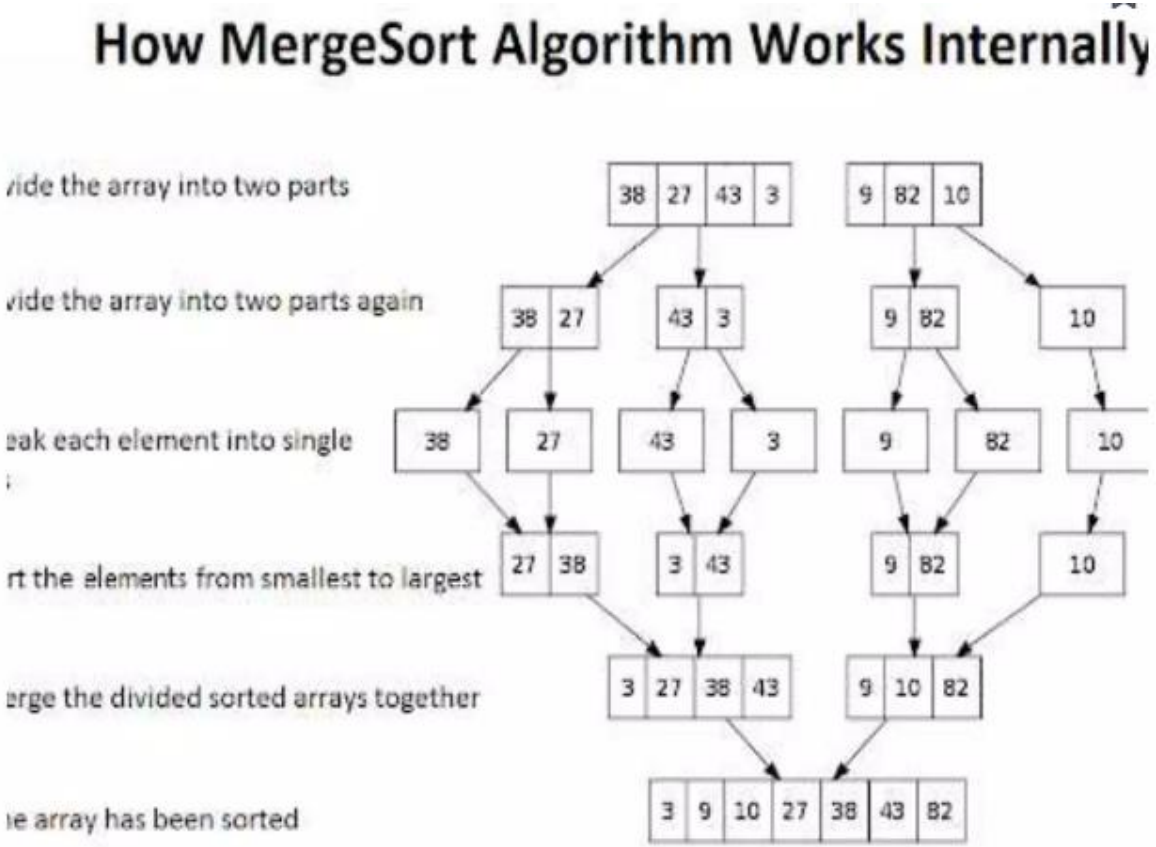
Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sub lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

Divide and conquer rule:-

A **divide-and-conquer** algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

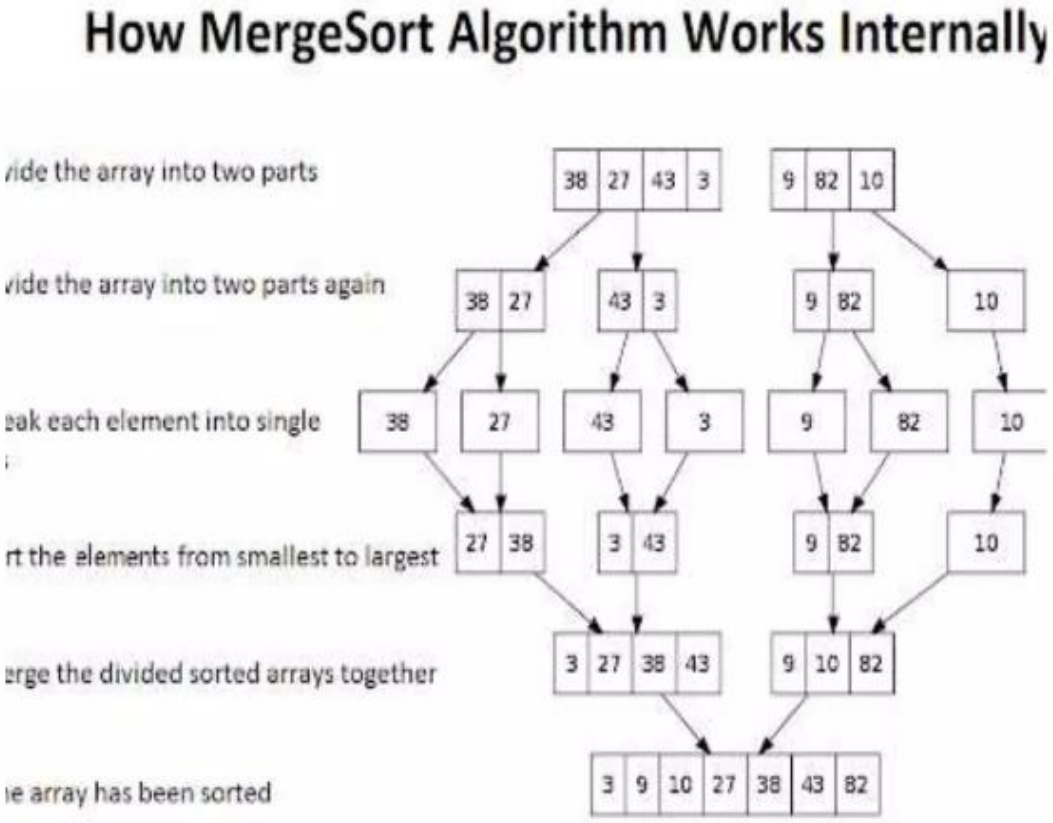
Algorithm(divide)

```
Mergesort(A,  lb  , ub,)
{
If(lb < ub)
{
Mid = lb+ub/2
Mergesort (A,lb,mid);
Mergesort (A,mid+1,ub);
Merge (A,lb,mid,ub)
}
}
```



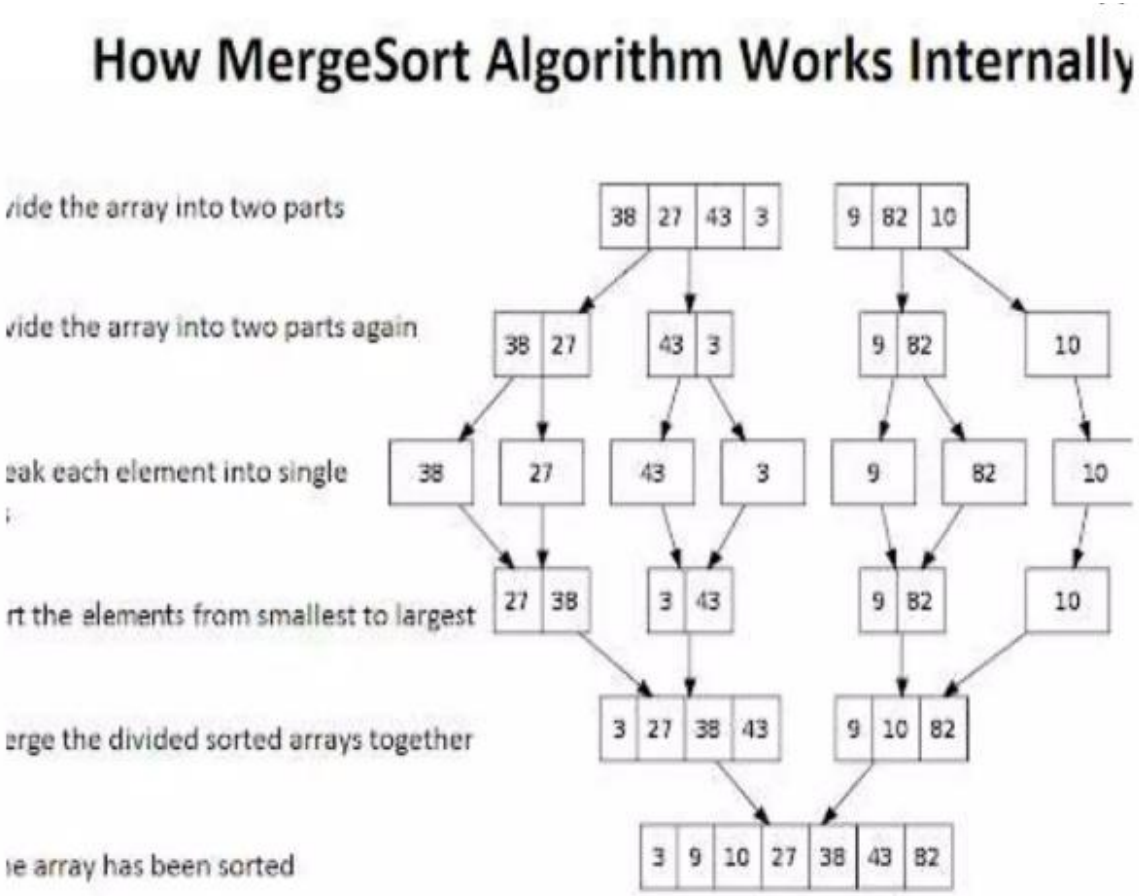
Algorithm(Merge)

```
Mergesort( A, lb , mid , ub)
{
i=lb;
J=mid +1;
K= lb;
While ( i<=mid && j<=ub)
{
If(a[i]<=a[j])
{
b[k]=a[i]
i++;
```



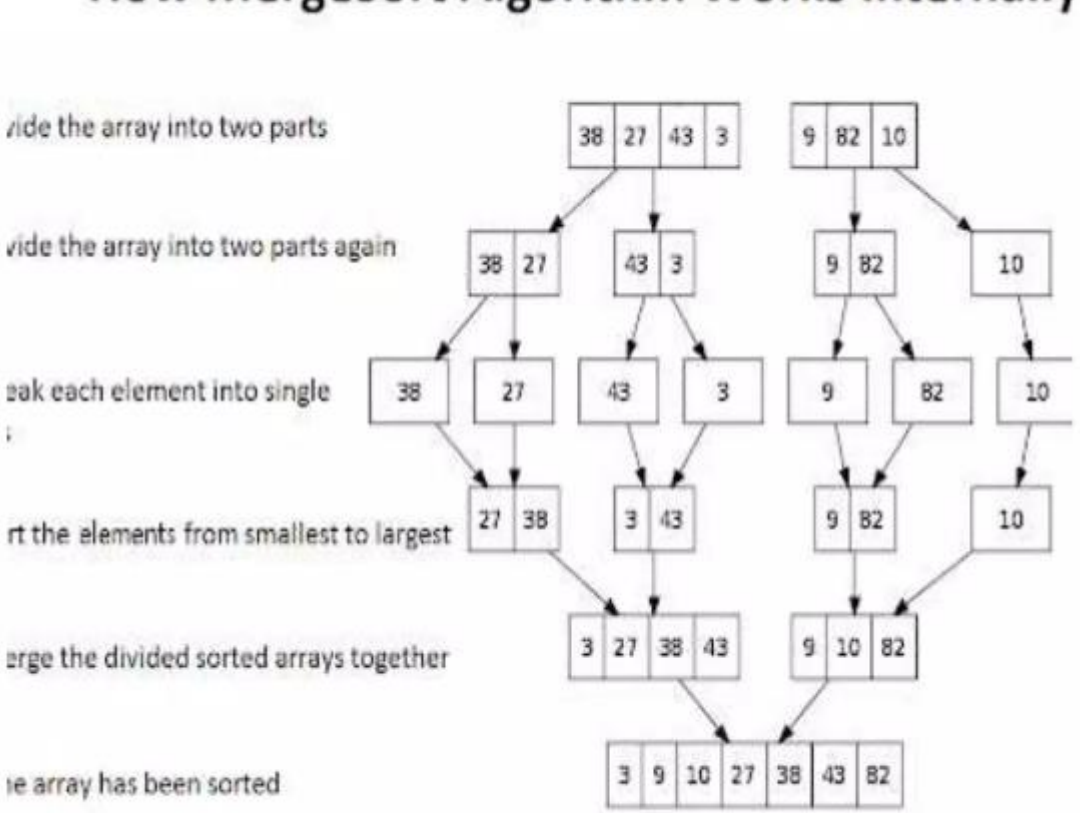
Algorithm(merge)

```
}
Else
b[k]=a[j] ;
J++;
}
K++;
}}
If(i>mid)
{ while (j< =ub)
B[k]=a[j];
J++;
K++;
```



Algorithm(merge)

```
}
Else
{
While (i<=mid)
B[k]=a[i];
i++;
K++;
}
}
For(k=lb; k <=ub ; k++)
{ a[k]= b[k];
}}
```



Quick Sort

- This sorting algorithm uses Divide and Conquer approach , the three-step divide-and-conquer process for sorting a array A[p….r]:
- **DIVIDE-**Partition the array A[p…r] into two subarrays A[p…q-1] and A[q+1..r] such that each element of left subarray is less than or equal to A[q] and each element of right subarray is greater than or equal to A[q] , for this partitioning we need to compute the index q.
- **Conquer-**Sort the two subarrays by recursive calls to quick sort.
- **Combine-**combine the two sorted subarrays
- Quick sort is inplace and not stable sorting algo.

Procedure

Quicksort(A, p, r)

1. if (p < r)
2. q=partition(A, p, r)
3. Quicksort(A, p, q-1)
4. Quicksort(A, q+1, r)///initial call is **Quicksort**(A,0,n-1) where n is no. of elements in array

partition(A, p, r)

1. x=A[r]
2. i = p-1
3. for j =p to r-1
4. if (A[j]≤x)
5. i = i+1
6. exchange A[i] with A[j]
7. exchange A[i+1] with A[r]
8. return i+1

Example

As i=p-1 so i=-1

x=4

2	8	7	1	3	5	6	4
p , j=0	1	2	3	4	5	6	r=7

2	8	7	1	3	5	6	4
p , i=0	j=1	2	3	4	5	6	r=7

2	8	7	1	3	5	6	4
p , i=0	1	j=2	3	4	5	6	r=7

2	8	7	1	3	5	6	4
p , i=0	1	2	j=3	4	5	6	r=7

2	1	7	8	3	5	6	4
0	i=1	2	3	j=4	5	6	r=7

2	1	3	8	7	5	6	4
p=0	1	i=2	3	4	j=5	6	r=7

2	1	3	8	7	5	6	4
p=0	1	i=2	3	4	5	j=6	7

2	1	3	8	7	5	6	4
p=0	1	i=2	3	4	5	6	7

Left sub array

right sub array

2	1	3	4(pivot)	7	5	6	8
p=0	1	i=2	3	4	5	6	7

Heap Sort

Heap Sort

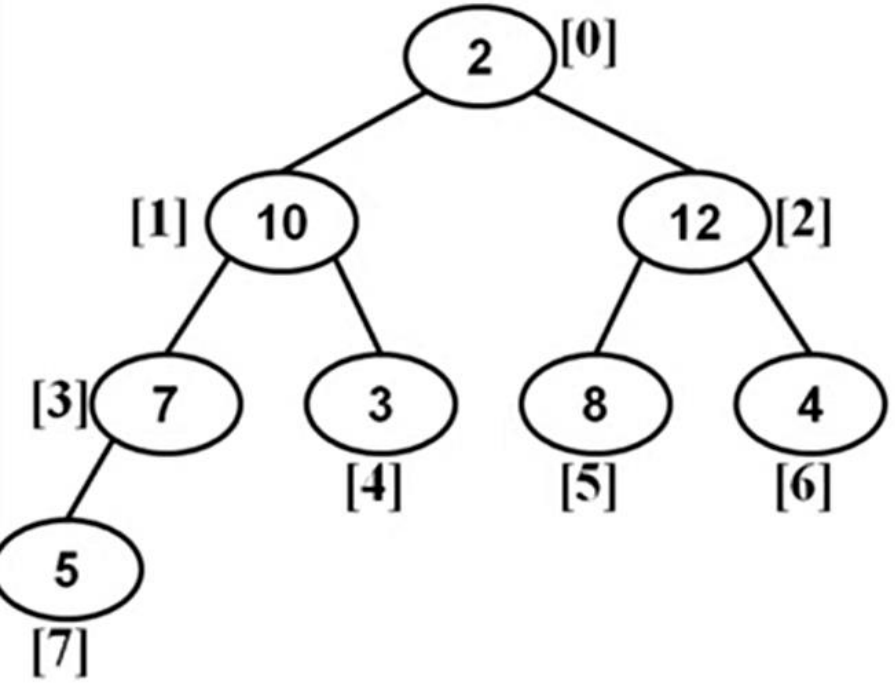
- **Heap**
  - A heap is a complete binary tree

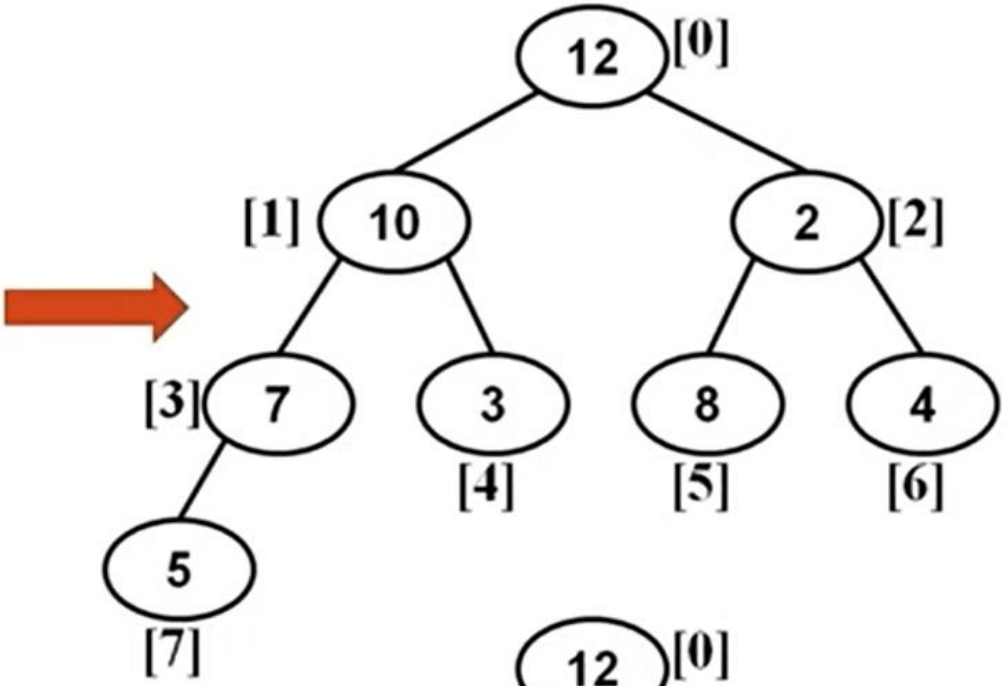
- **Heap Sort**
  - Heap sort is a popular and efficient sorting algorithm.
  - The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list

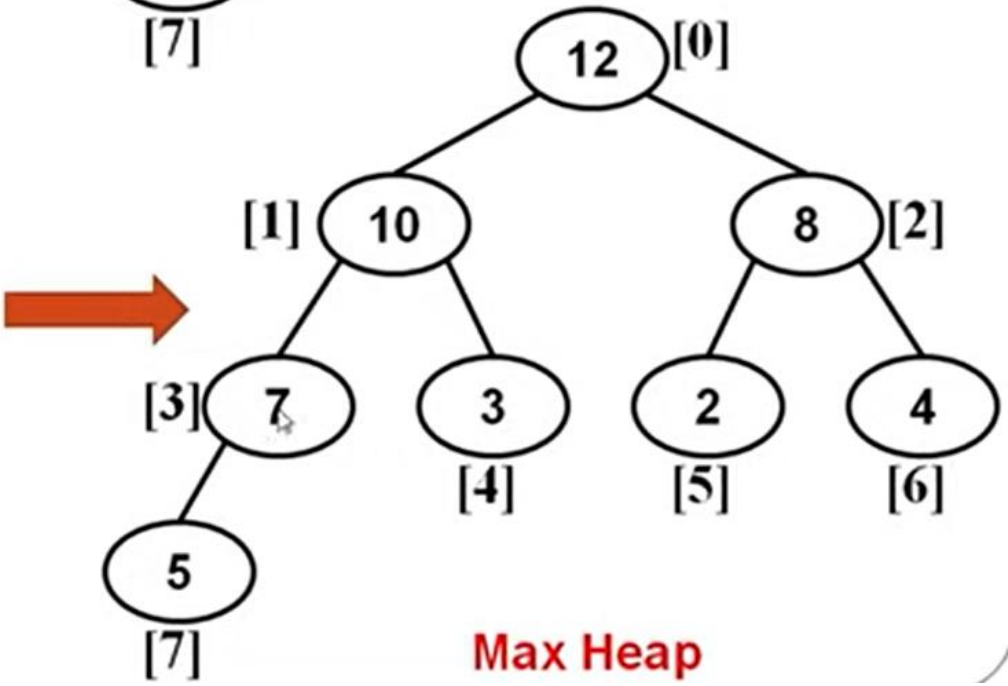
Heap Sort Algorithm

- **Steps**
  1. The first step includes the creation of a max heap by adjusting the elements of the array.
  2. After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

- Apply Heapify on the element at index 0







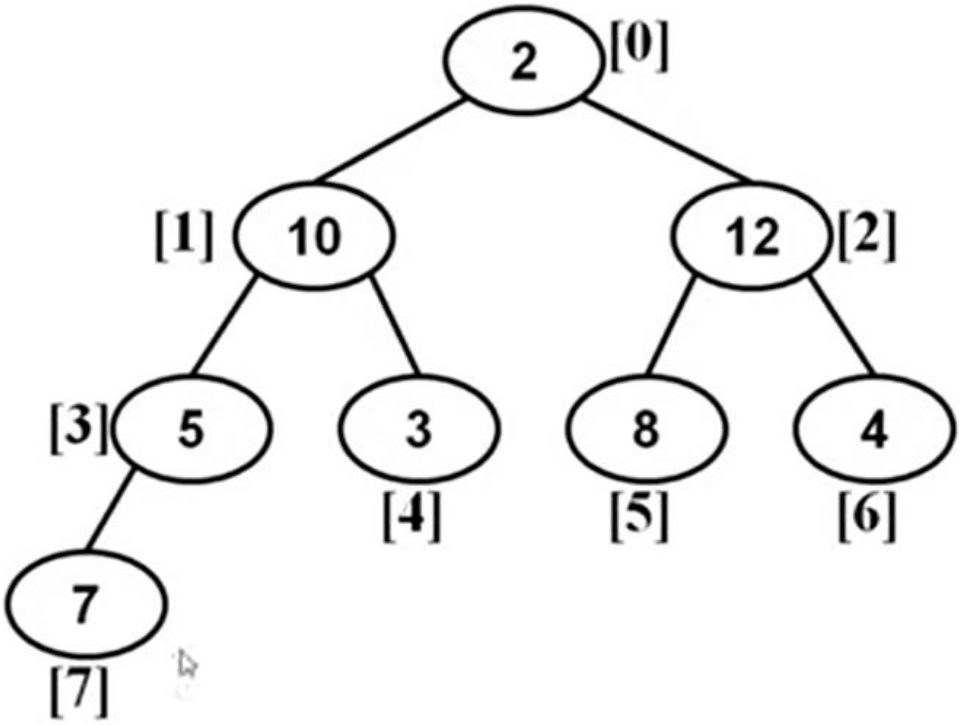
Max Heap

Heap Sort Example

- Consider the array

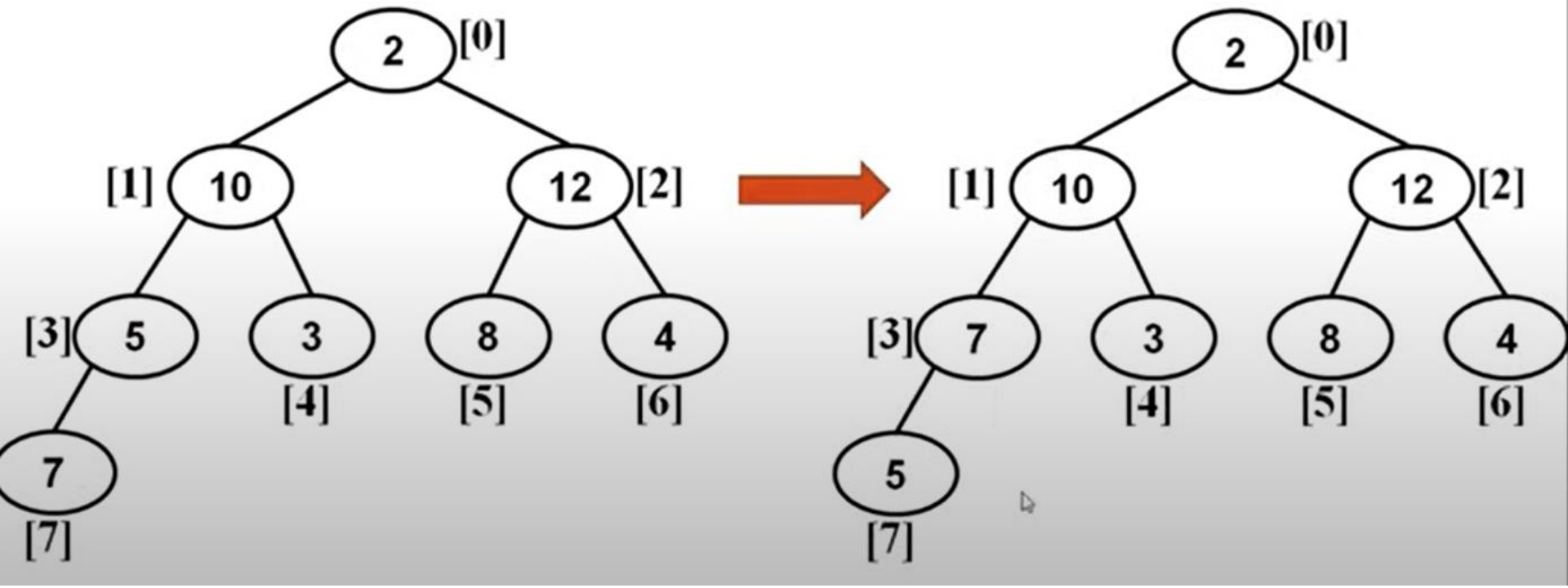
2	10	12	5	3	8	4	7
0	1	2	3	4	5	6	7

- Corresponding complete binary tree

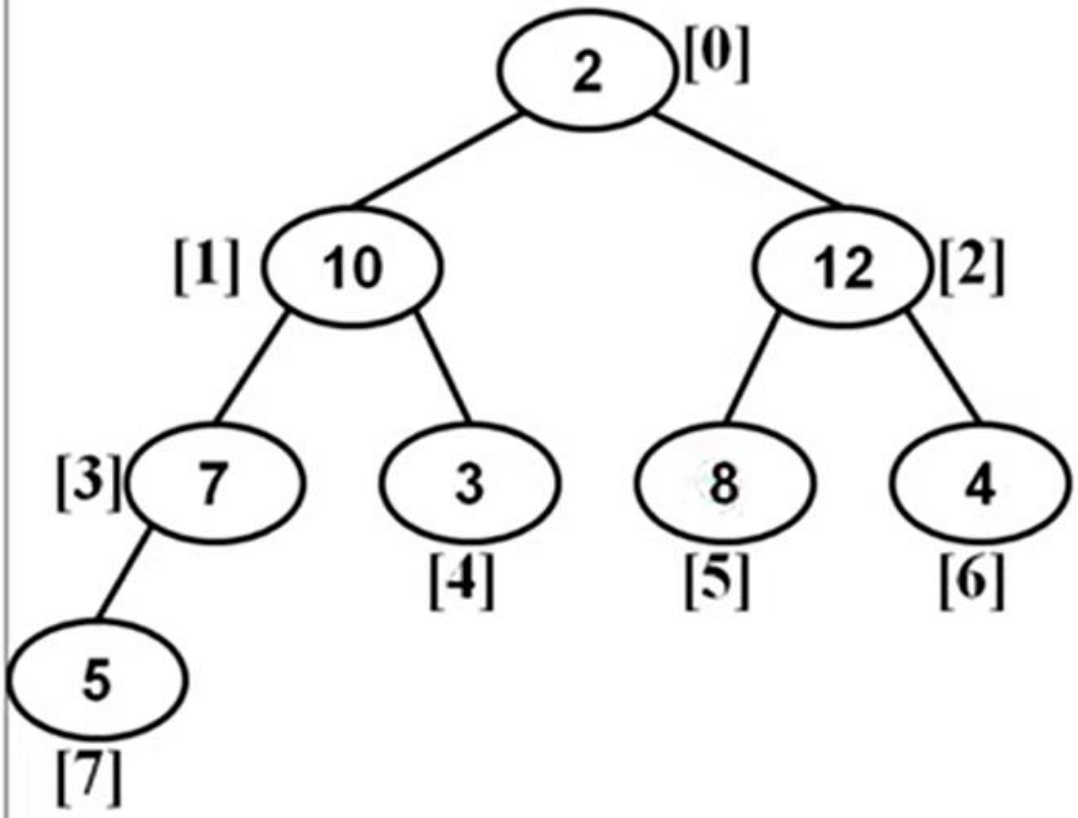


- Apply Heapify on the element at index n/2-1

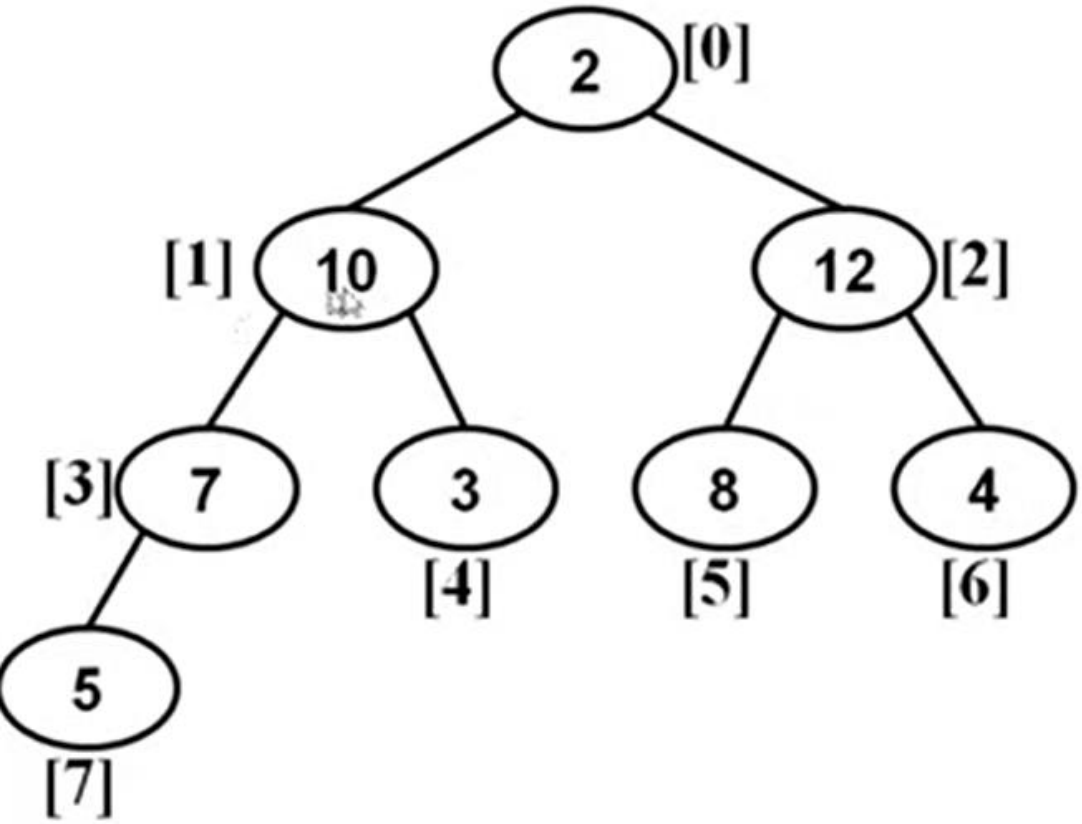
=3



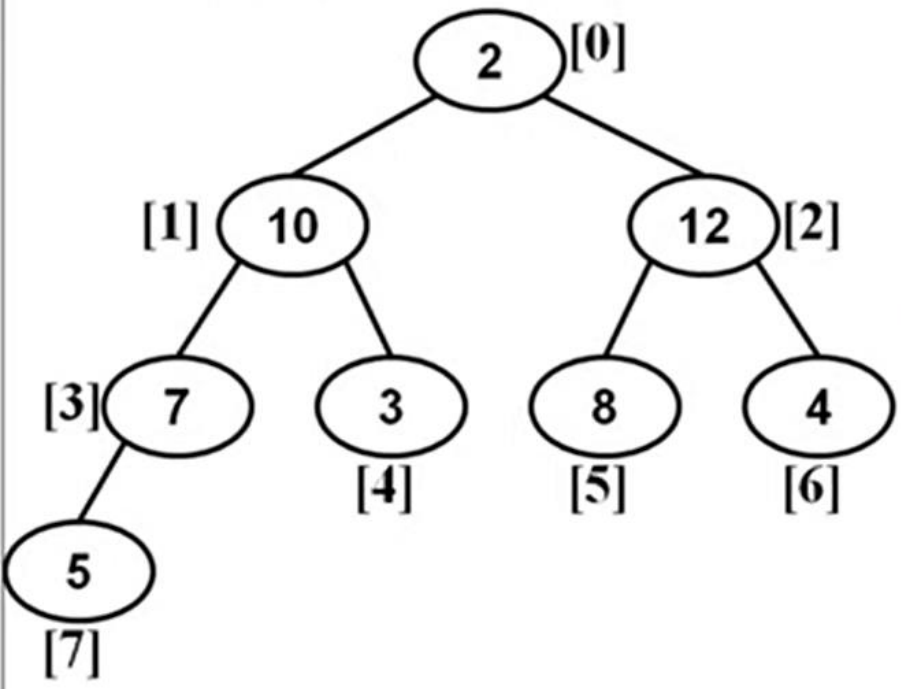
- Apply Heapify on the element at index 2

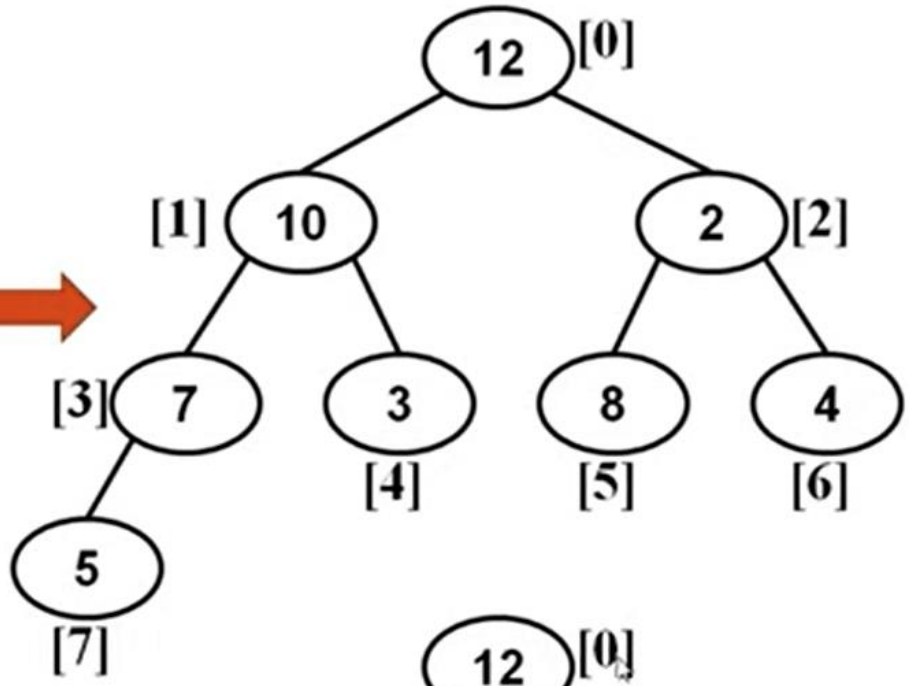


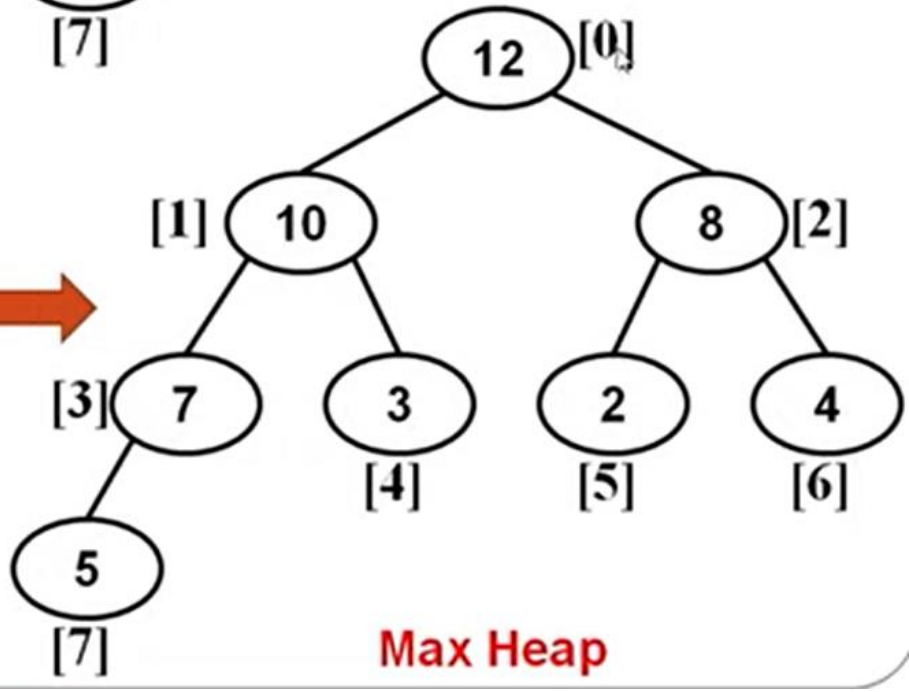
- Apply Heapify on the element at index 1



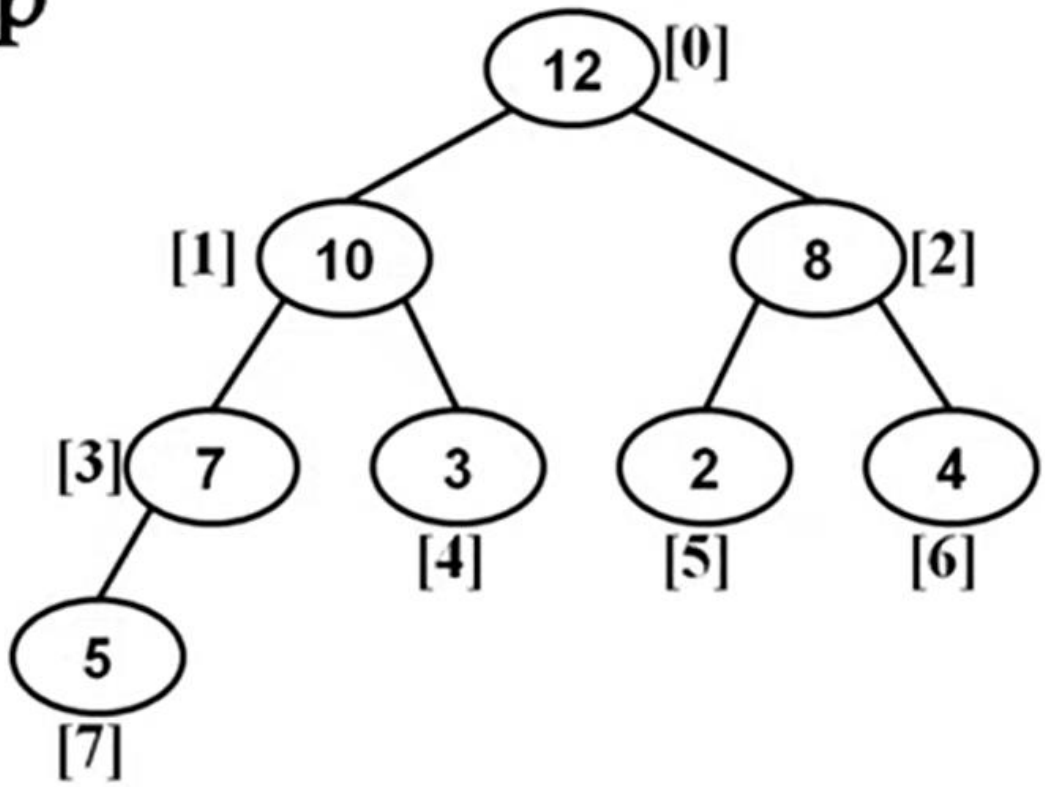
- Apply Heapify on the element at index 0







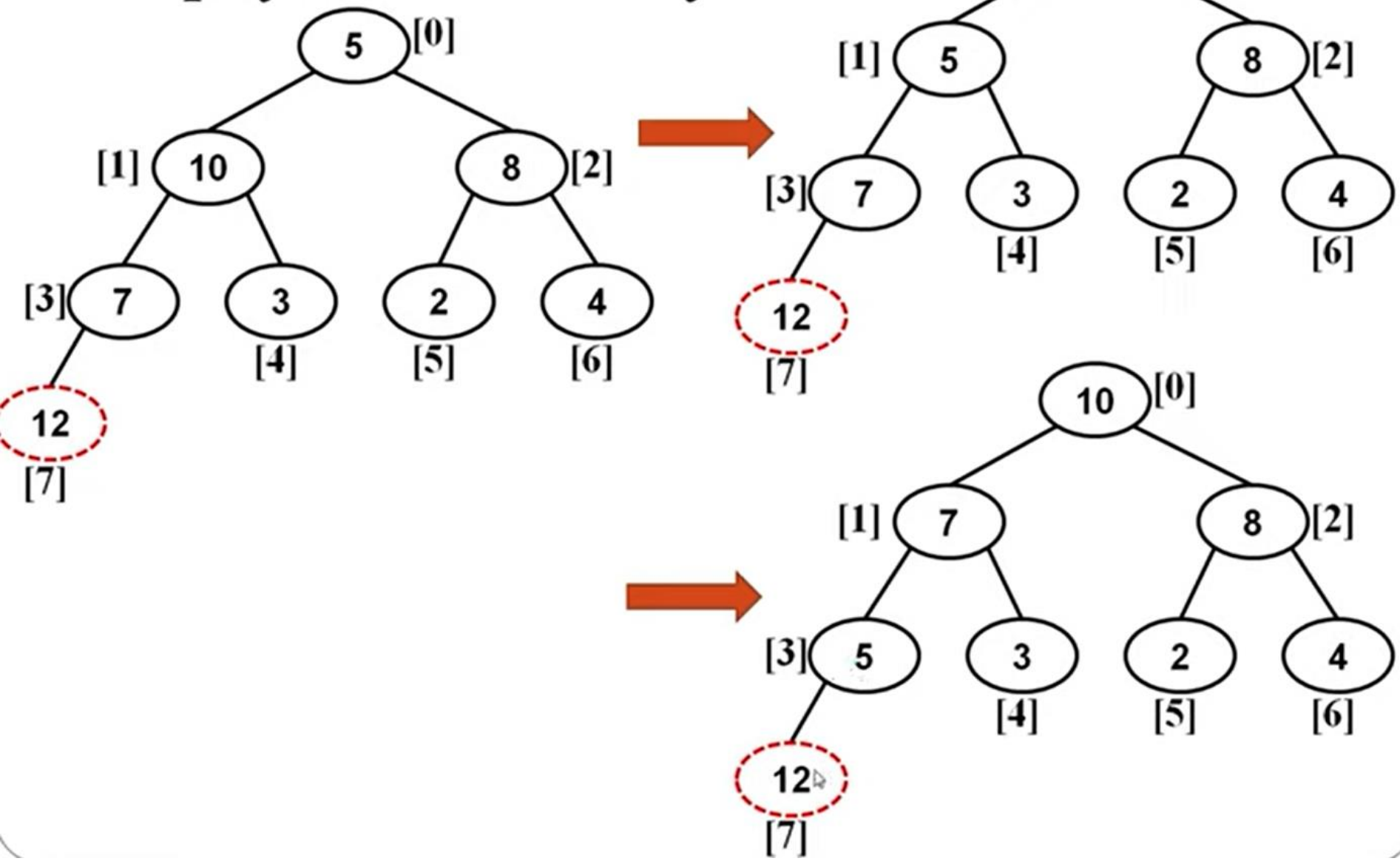
- **Resultant Heap**



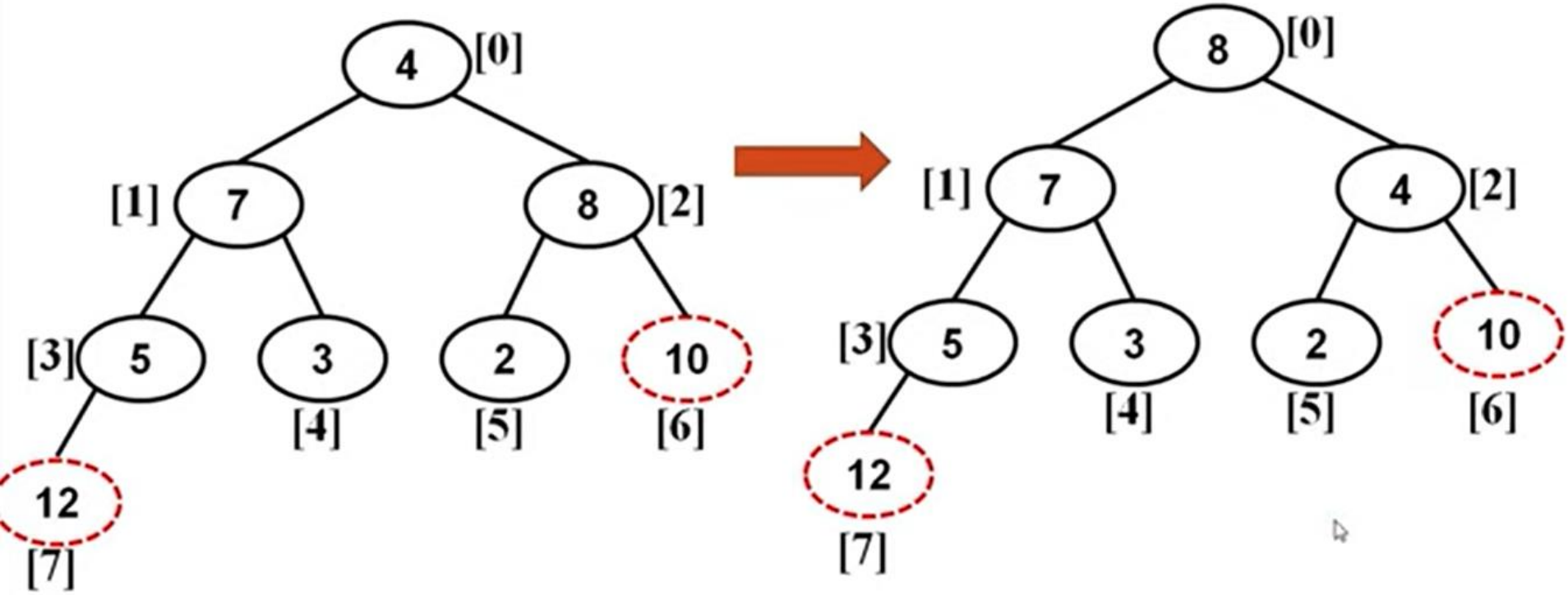
- **Corresponding array**

12	10	8	7	3	2	4	5
0	1	2	3	4	5	6	7

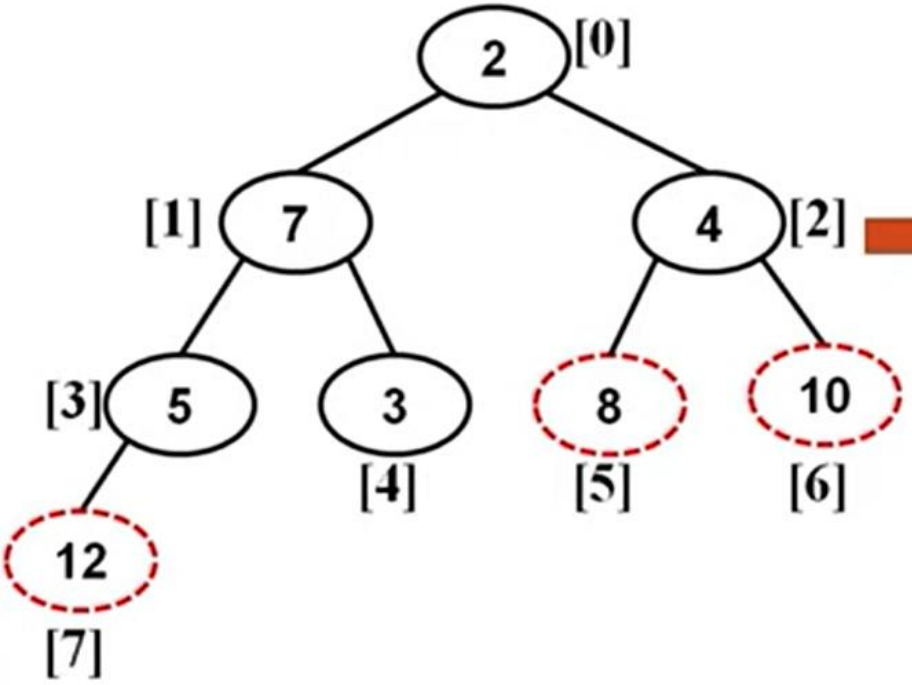
- Swap A[0] and A[7]
- Heapify A[0] with array size 7

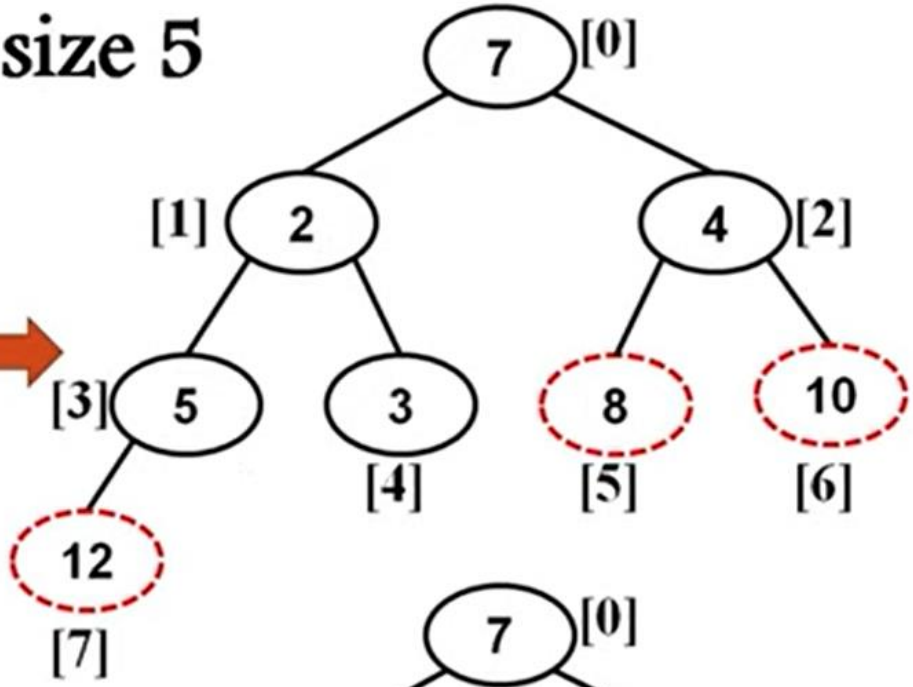


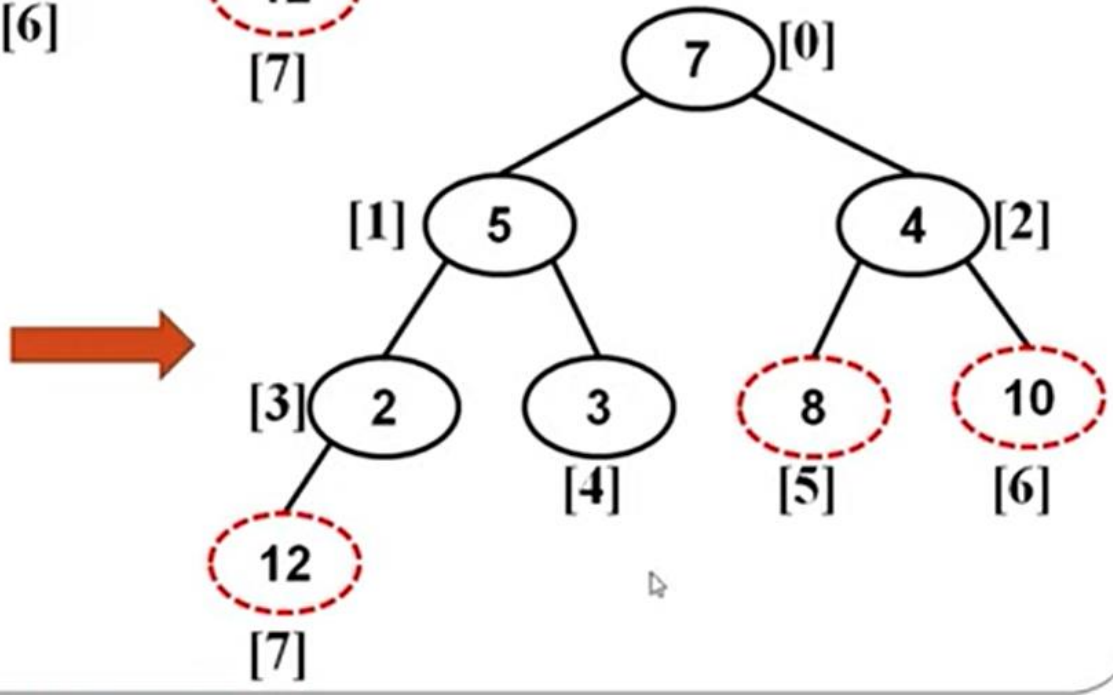
- Swap A[0] and A[6]
- Heapify A[0] with array size 6

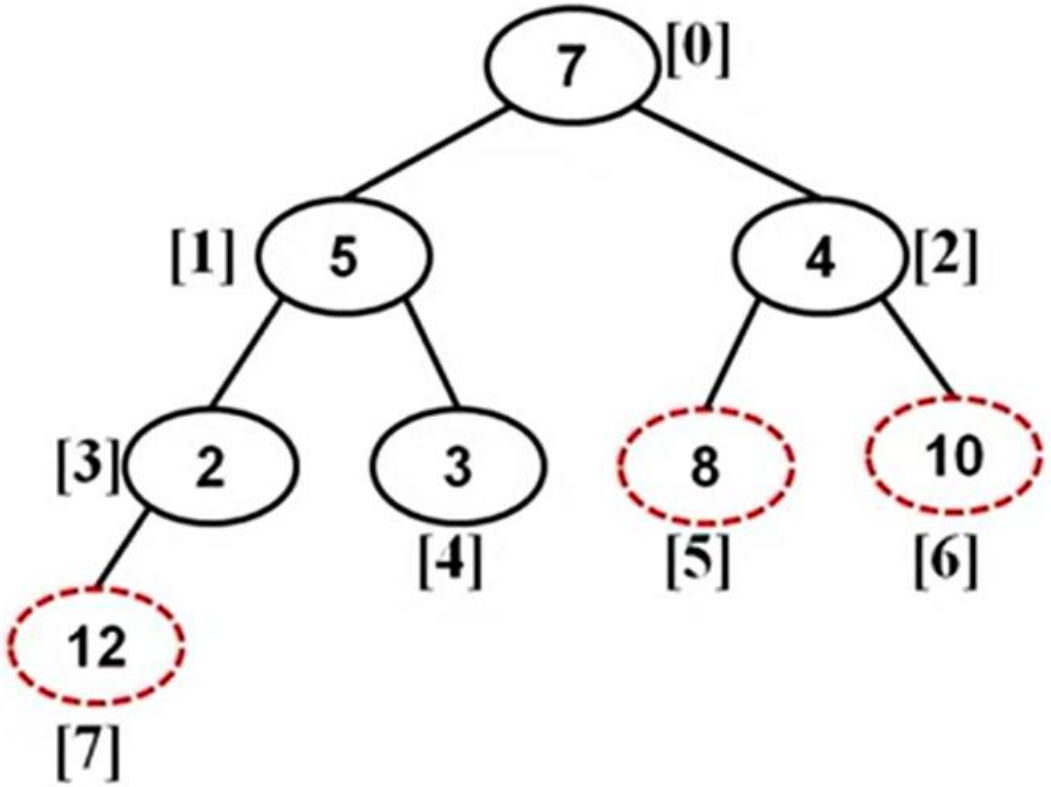


- Swap A[0] and A[5]
- Heapify A[0] with array size 5





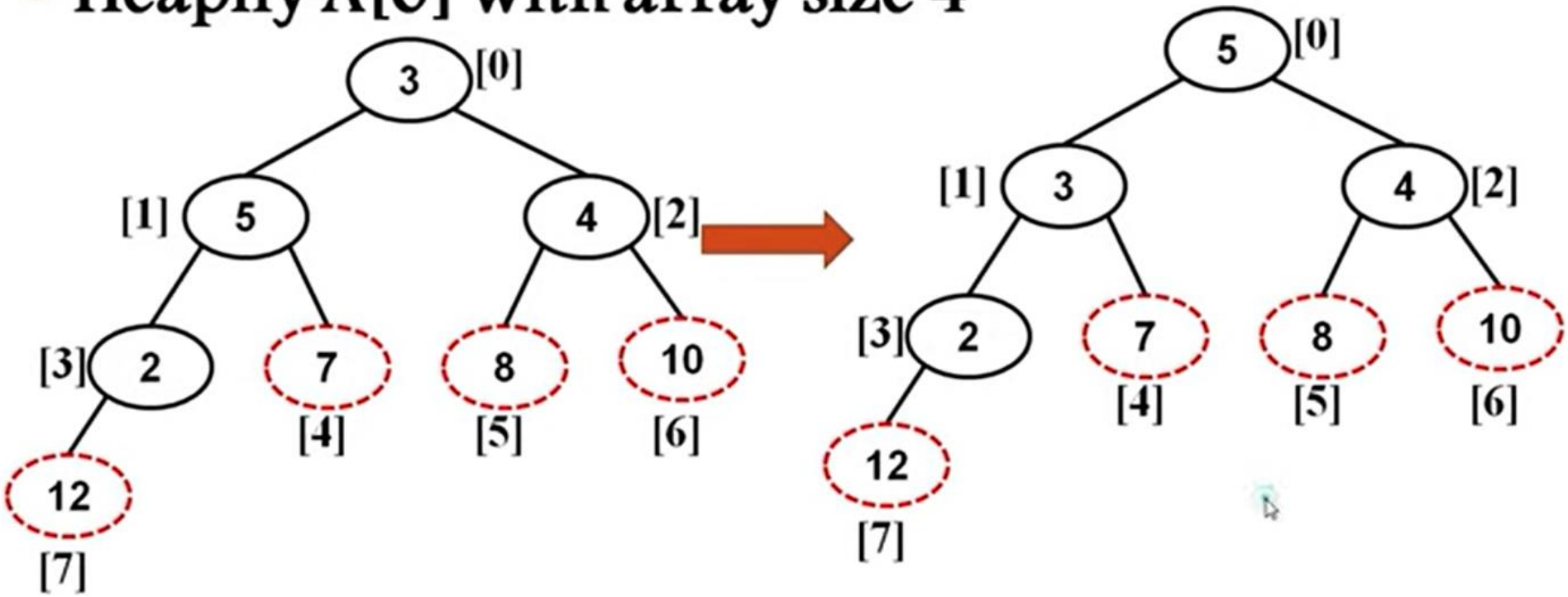


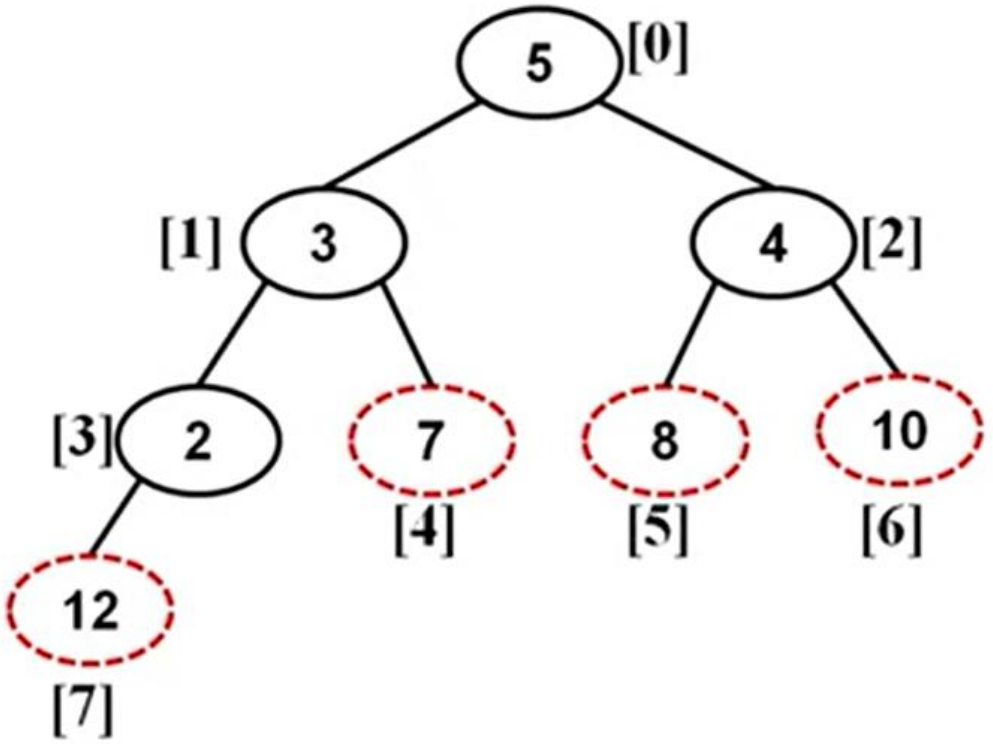


- Corresponding array

7	5	4	2	3	8	10	12
0	1	2	3	4	5	6	7

- Swap A[0] and A[4]
- Heapify A[0] with array size 4

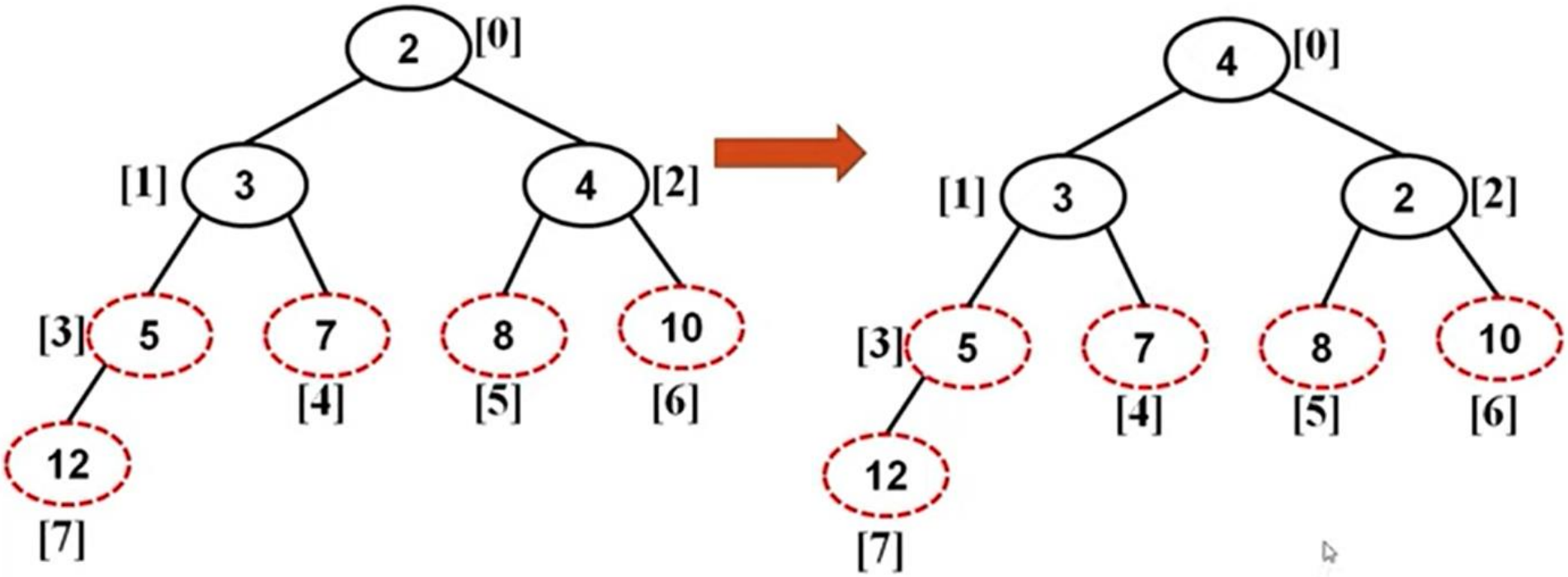




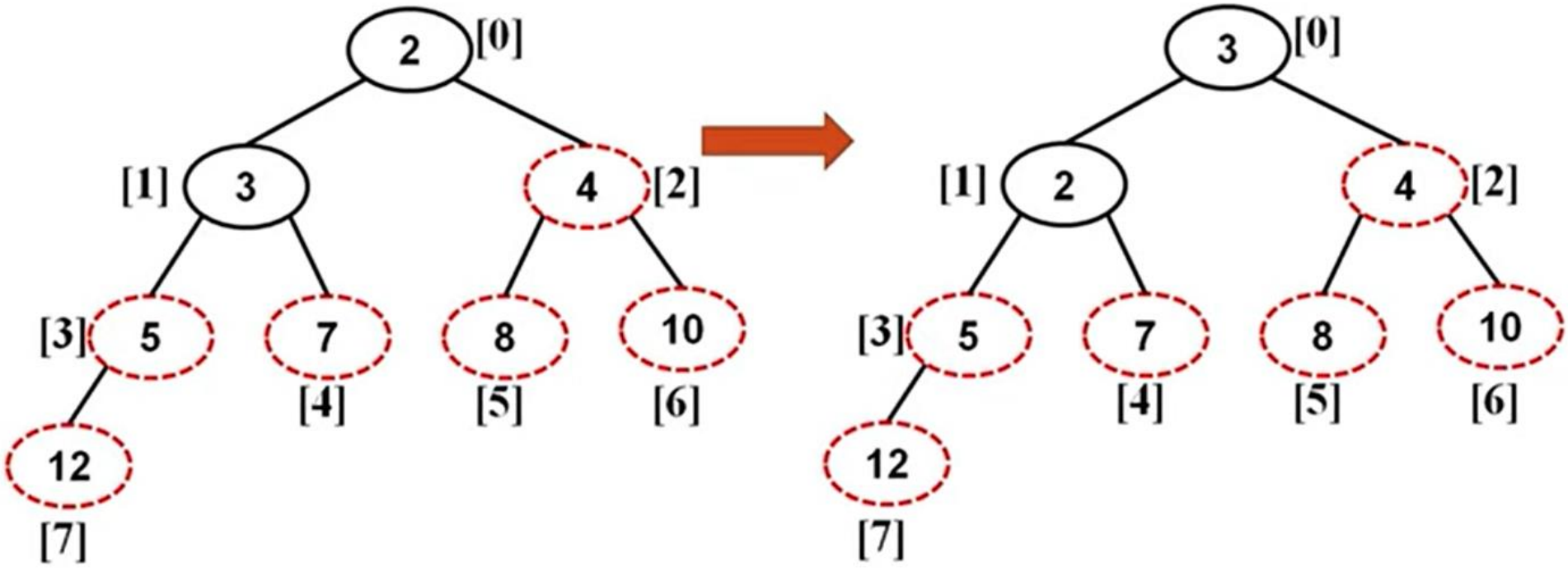
- Corresponding array

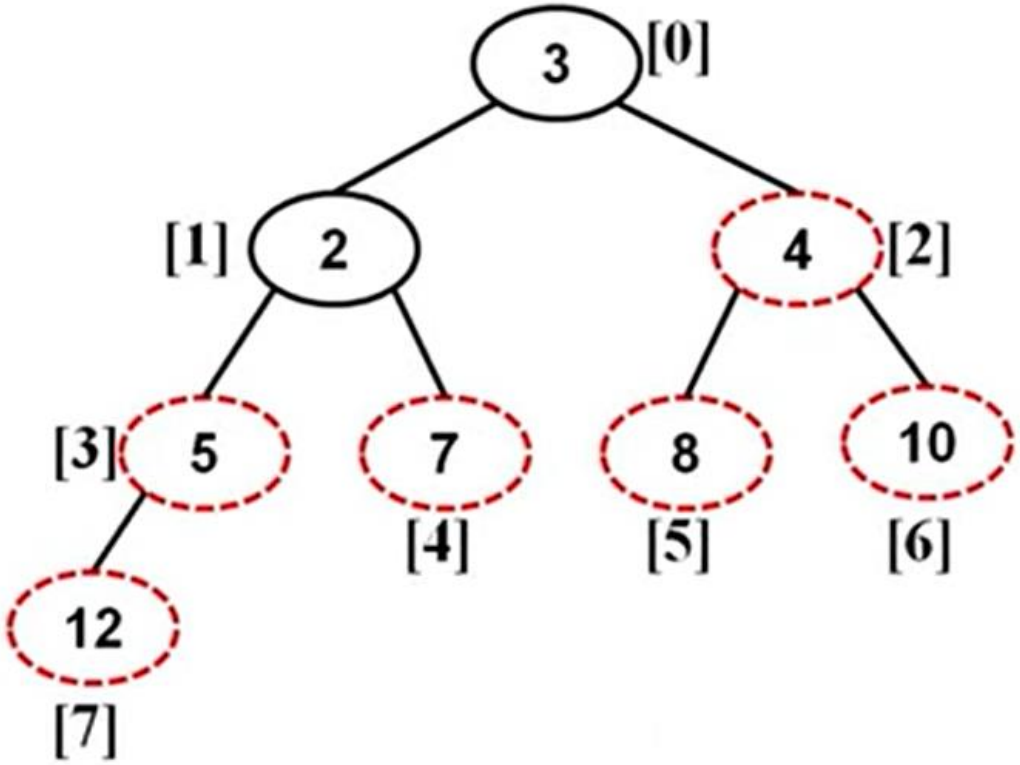
5	3	4	2	7	8	10	12
0	1	2	3	4	5	6	7

- Swap A[0] and A[3]
- Heapify A[0] with array size 3



- Swap A[0] and A[2]
- Heapify A[0] with array size 2

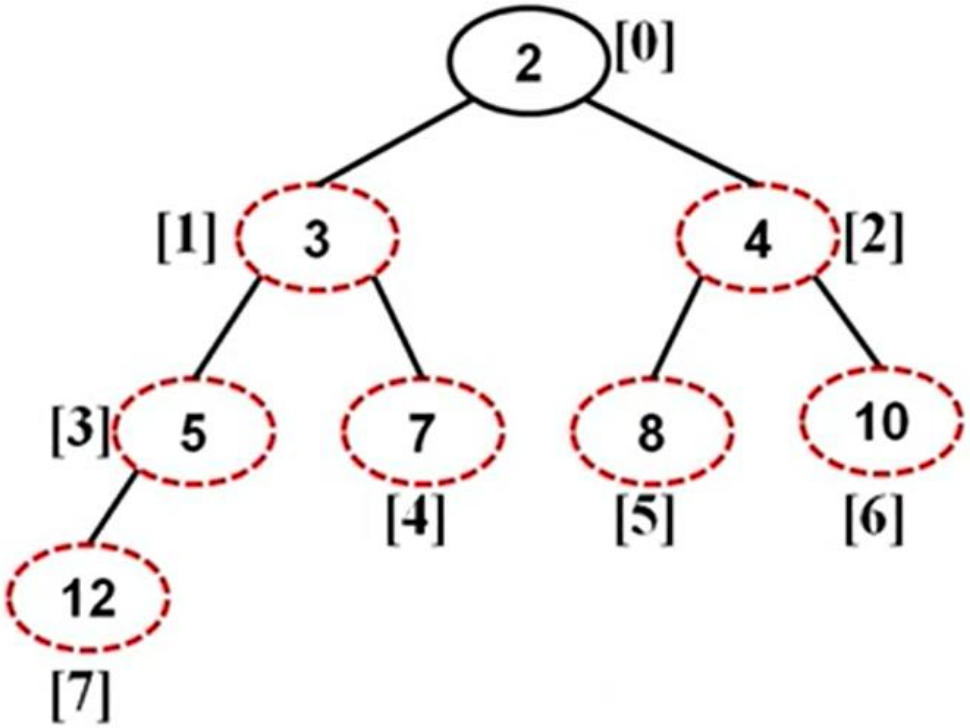




- Corresponding array

3	2	4	5	7	8	10	12
0	1	2	3	4	5	6	7

- Swap A[0] and A[1]



- Corresponding array

2	3	4	5	7	8	10	12
0	1	2	3	4	5	6	7

Heap Sort Algorithm

```
Algorithm HeapSort(A, n)
1. BuildMaxHeap(A,n)
2. for i = n-1 to 0
   1. swap A[0] with A[i]
   2. Heapify(A,i,0)
```

```
Algorithm BuildMaxHeap(A,n)
1. for i = n/2-1 to 0
   1. Heapify(A,n,i)
```

```
Algorithm Heapify(A,n,i)
1. L = LeftChildIndex(i)
2. R = RightChildIndex(i)
3. largest=i
4. if L<n and A[L] > A[largest]
   1. largest = L
5. if R < n and A[R] > A[largest]
   1. largest = R
6. if largest != i
   1. swap A[i] with A[largest]
   2. Heapify(A,n,largest)
```

Introduction to Hashing

Hash Table

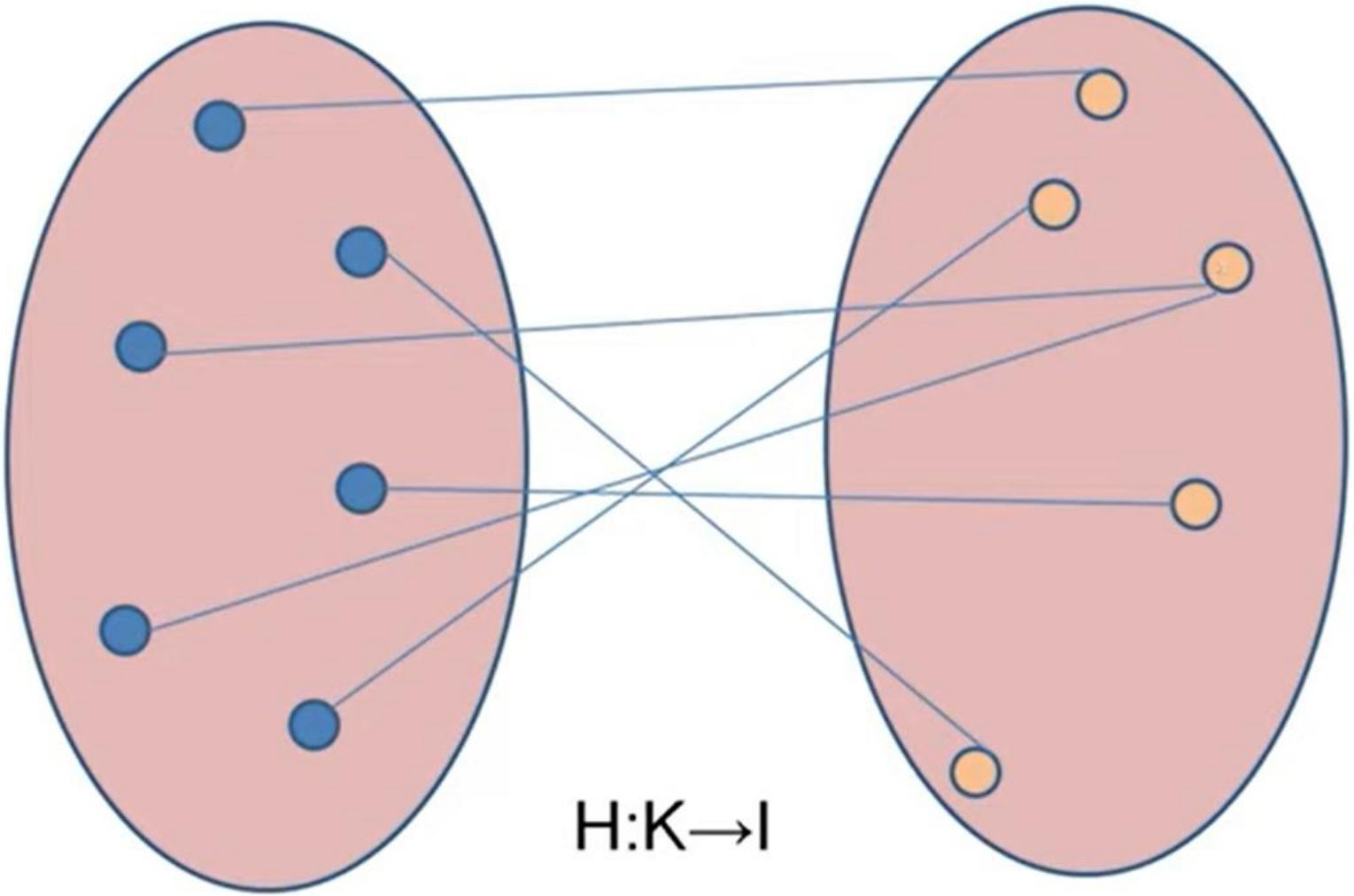
- Hash Table is a data structure which help us to retrieve information very effectively.
- In hash table, data is stored in array format where each data values has its own unique index value.
- Access of data becomes very fast if we know the index of desired data.
- Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data.
- Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from

Hashing

- Hashing is a technique of mapping keys into the hash table by using a hash function

- H: K⟶I where H is the mapping hash function
  K is the set of key value
  I is the range of indices
- The main idea behind any hashing technique is to find a one-to-one correspondence between a key value and index in the hash table where the key value can be placed.

Hashing



Hashing

- Few criteria while deciding hash function:
  - The function H should be very easy and quick to compute
  - The function H should as far as possible give two different indices for two different key values. There should be minimum no. of collision

Hash Functions

- Some popular hash functions are:
  1. Division method
  2. Mid square method
  3. Folding method
  4. Digit Analysis method

Hash Functions

1. **Division method**
   - The hash function H is
     - $H(k) = k \% m$

k is the key value

m is the size of the hash table

m is usually a prime number or a number without small divisors. This will minimize the number of collisions

- Example: k=102 and m=23

$$H(k) = k \% m = 102 \% 23 = 10$$

Hash Functions

1. **Division method**
   - **Advantages**: Fastest and easiest hash function
   - **Disadvantage**: Have to avoid certain value of m
   - **Good choice for m** :
     - Prime numbers
     - Not too close to power of 2 or 10

- Example: Assume that the hash table size is 8. Consider the key values 10,21,4,41,24,14,17,15,31. Apply division method and find the index of each key value.

- Ans: Here the hash function is H(k) = k%8

Key	Hash Function	Index
10	10%8	2
21	21%8	5
4	4%8	4
41	41%8	1
24	24%8	0
14	14%8	6
17	17%8	1
15	15%8	7
31	31%8	7

Hash Table

Index	Key	
0	24	
1	41,17	Collision
2	10	
3		
4	4	
5	21	
6	14	
7	15,31	Collision

Hash Functions

2. **Mid square method**
   - The hash function H is
     - $H(k) = x$
   - x is obtained by extracting some digits from the middle of k^2

k	k^2	H(k)	
65	4225	22	Extract 2 digits
90	8100	10	
150	22500	50	

Hash Functions

3. **Folding method**

- Here the key k is partitioned in to number of parts, $k_1, k_2 \ldots k_r$
- Each part except possibly the last, has the same no of digits as the required address.
- These parts are added together ignoring the last carry.
- The hash function $H(k) = k_1 + k_2 + \ldots K_r$

k	H(k)	Index
3407	34+07 = 41	41
9020	90+20=110	10
15887	15+88+9=112	12

Hash Functions

4. **Digit Analysis method**
   - Here, the hash address is formed by extracting the digits and rearranging them.
   - Example: key-6732541 can be transformed to hash function 427 by extracting digits in even positions and then reversing the combination.
   - The decision for extraction and rearrangement is based on some analysis.
   - Here, an analysis is performed to determine which key position should be used in forming hash address.
   - For each criterion, hash addresses are calculated and then a graph is plotted, then that criterion is selected which produces the most uniform distribution