Trees

Data Structures and Algorithm

Trees

Siblings

Degree of node

Successor node

Children

Parent node

Root

Depth

Internal node

Leaf node

Height

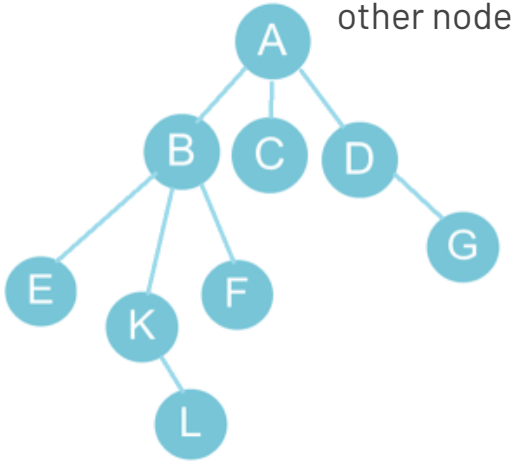
Path

Ancestor

Descendant

Trees

- Extend the concept of linked data structure to a structure that may have multiple relations among its nodes Such a structure is called a **tree**.
- A tree is a collection of nodes connected by directed (or undirected) edges.
- A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more sub trees. A tree has following general properties:
  - One node is distinguished as a **root**;
  - Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent -> children*



A is a parent of B, C, D,
B is called a child of A.
on the other hand, B is a parent of E, F, K
In the given picture, the root has 3 subtrees.

Advantages of Trees

Trees are so useful and frequently used, because they have some very serious advantages:

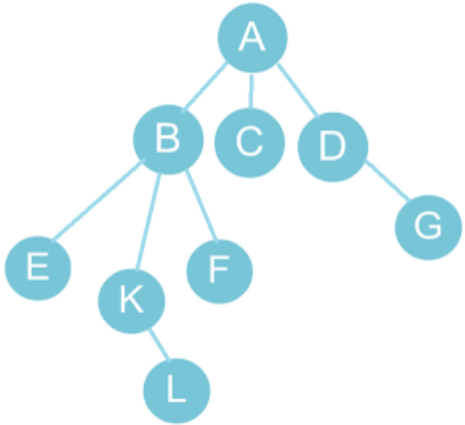
- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minumum effort.

Trees

- A rooted tree data structure stores information in *nodes*
  - Similar to linked lists:
    - There is a first node, or *root*
    - Each node has variable number of references to **successors**
    - Each node, other than the root, has exactly one node pointing to it
    - All nodes will have zero or more **child nodes** or *children*
    - For all nodes other than the root node, there is one parent node

Trees

- Each node can have *arbitrary* number of children. Nodes with no children are called **leaves**, or **external** nodes. In the above picture, C, E, F, L, G are leaves. Nodes, which are not leaves, are called **internal** nodes. Internal nodes have at least one child.
- Nodes with the same parent are called **siblings**. In the picture, B, C, D are called siblings. The **depth of a node** is the number of edges from the root to the node. The depth of K is 2. The **height of a node** is the number of edges from the node to the deepest leaf. The height of B is 2. The **height of a tree** is a height of a root from deepest node.



A is a parent of B, C, D,
B is called a child of A.
on the other hand, B is a parent of E, F, K

Trees

All nodes will have zero or more **child nodes** or *children*

- I has three children: J, K and L

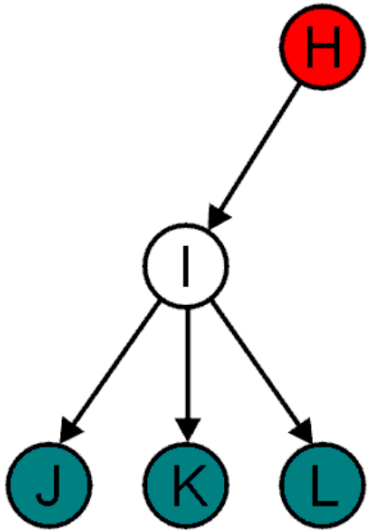
For all nodes other than the root node, there is one parent noc

- H is the parent

The ***degree of a node*** is defined as the number of its children:

Nodes with the same parent are *siblings*

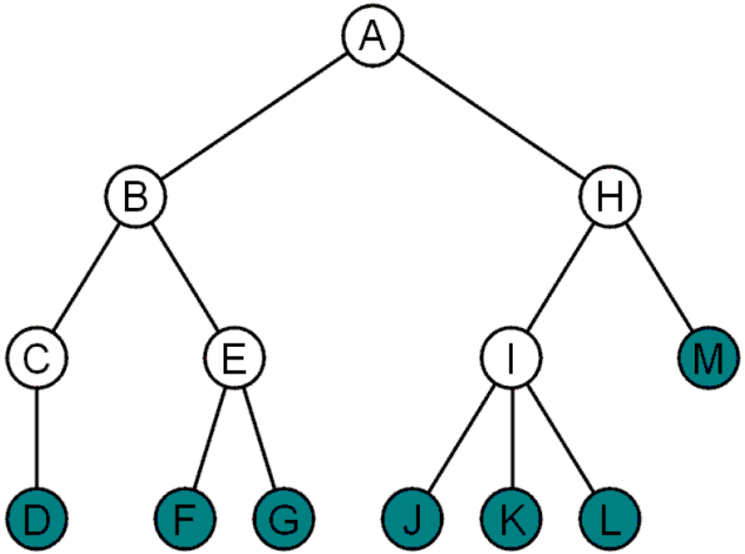
- J, K, and L are siblings



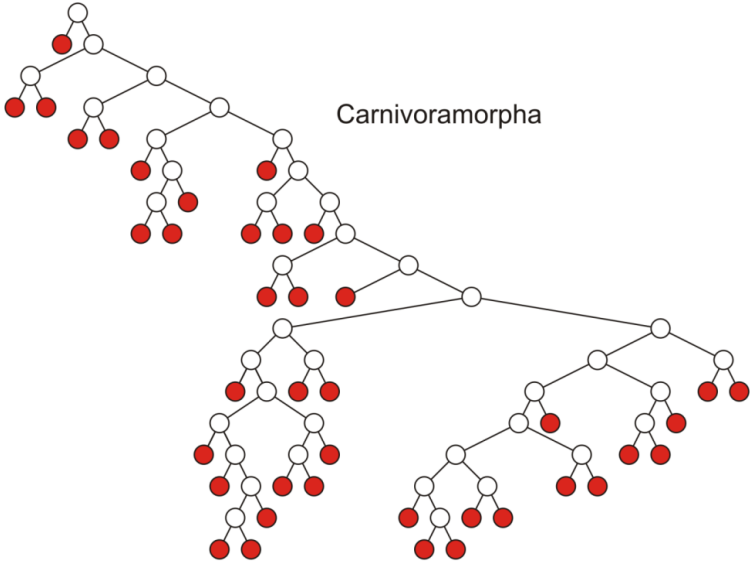
Trees

Nodes with degree zero are also called *leaf nodes*

All other nodes are said to be *internal nodes*, that is, they are internal to the tree

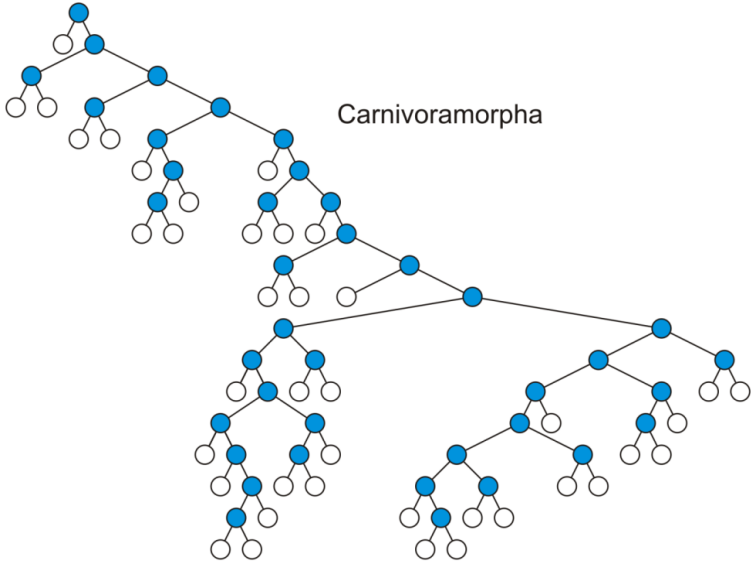


Tree



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Leaf nodes:



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Internal nodes:

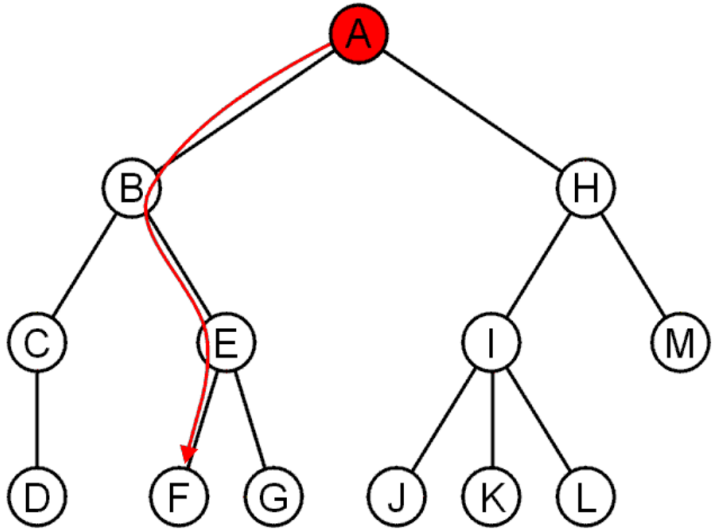
Trees

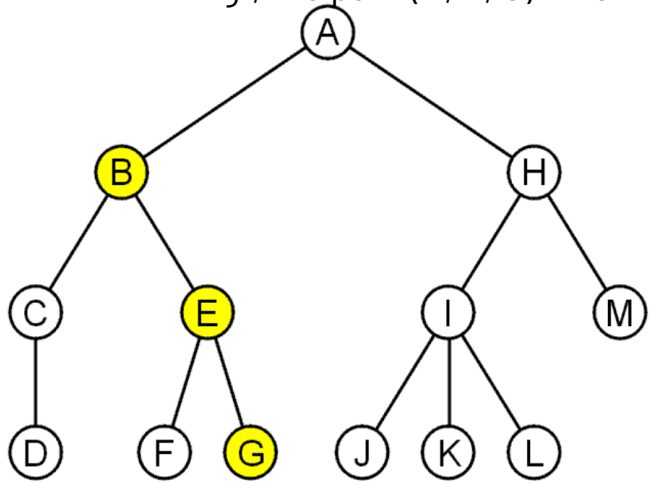
- The shape of a rooted tree gives a natural flow from the *root node*, or just *root*

A path is a sequence of nodes
$(a_0, a_1, \ldots, a_n)$
where a_{k+1} is a child of a_k is

The length of this path is n

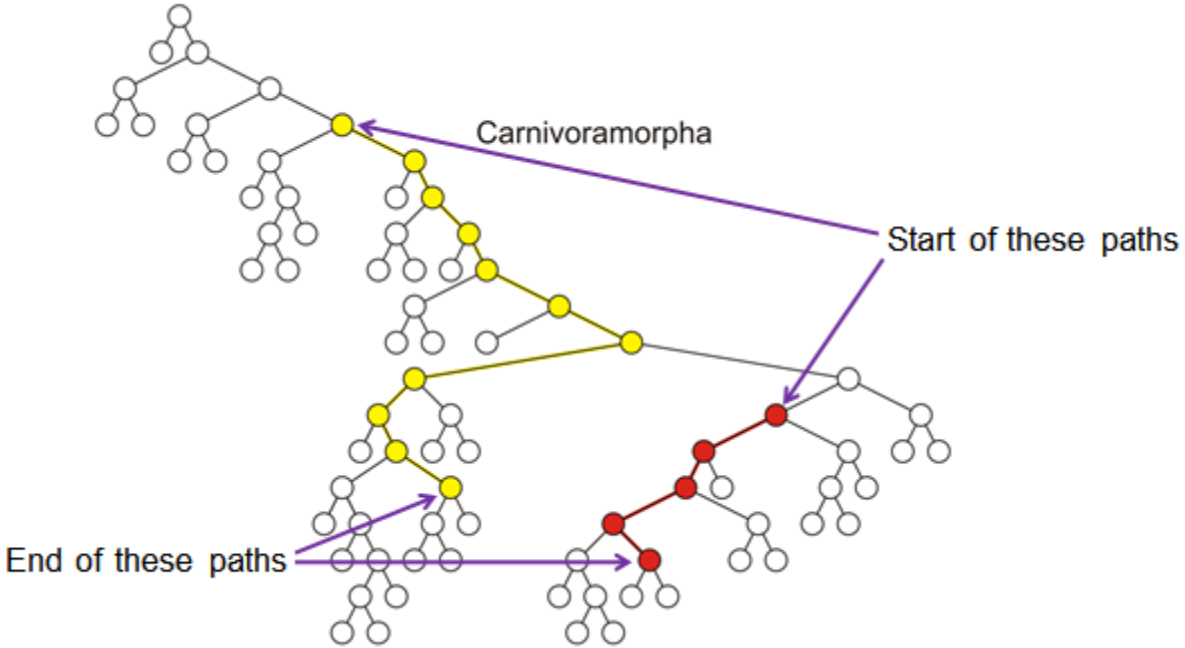
E.g., the path (B, E, G) has length 2





Trees

- Paths of length 10 (11 nodes) and 4 (5 nodes)

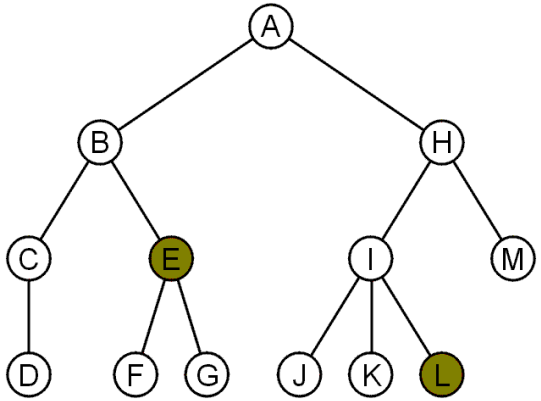


Trees

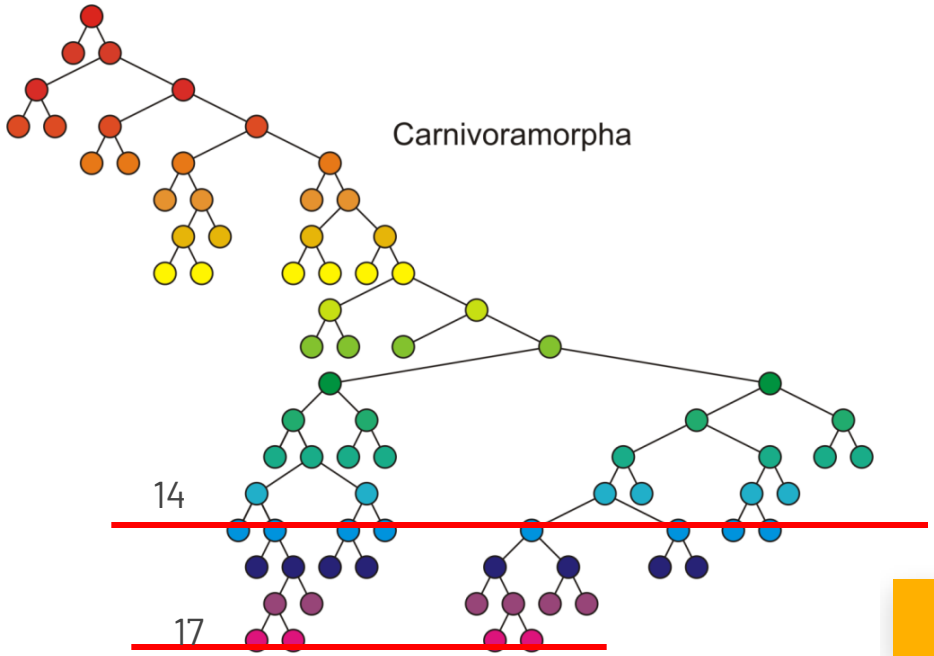
For each node in a tree, there exists a unique path from the root node to that node

The length of this path is the *depth* of the node, *e.g.*,

- E has depth 2
- L has depth 3



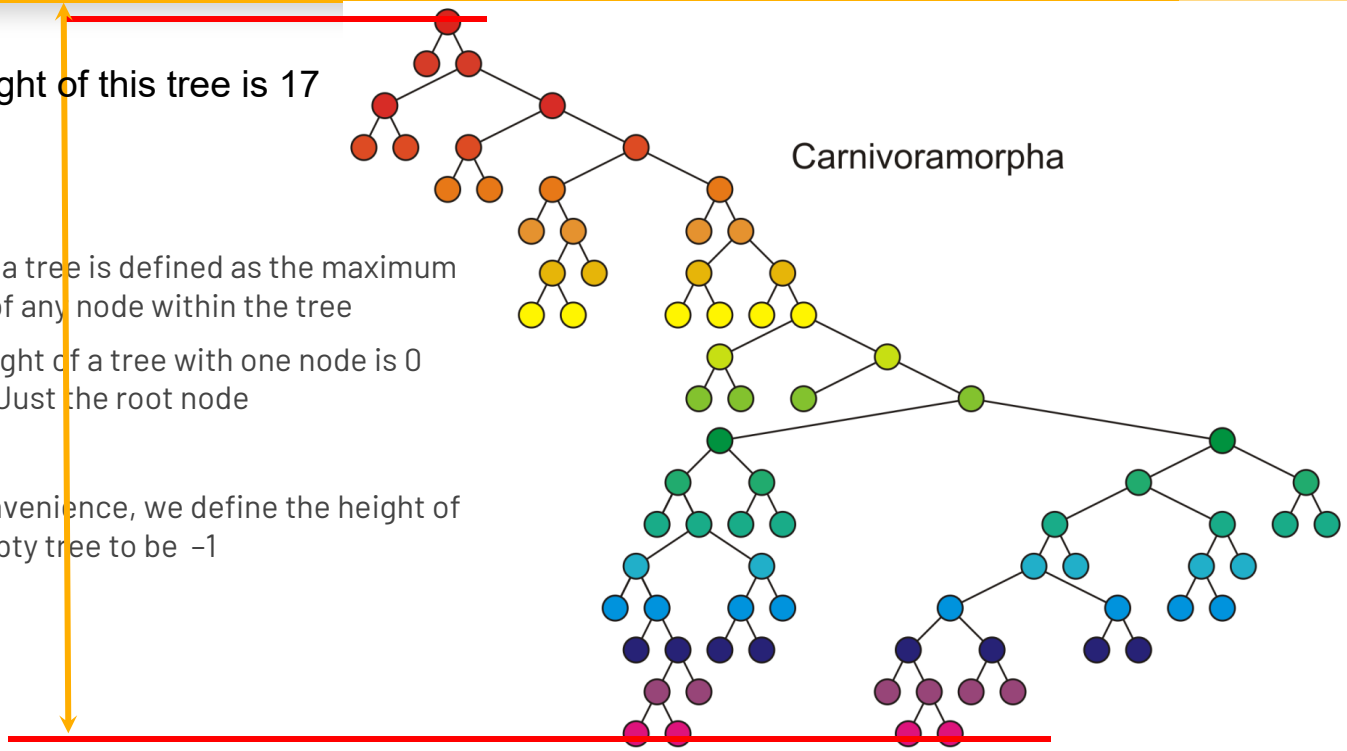
Nodes of depth up to 17



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Trees

The height of this tree is 17



The *height* of a tree is defined as the maximum depth of any node within the tree

The height of a tree with one node is 0

- Just the root node

For convenience, we define the height of the empty tree to be –1

Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Trees

If a path exists from node *a* to node *b*:

- *a* is an *ancestor* of *b*
- *b* is a *descendent* of *a*

Thus, a node is both an ancestor and a descendant of itself

- We can add the adjective *strict* to exclude equality: *a* is a *strict descendent* of *b* if *a* is a descendant of *b* but $a \neq b$

The root node is an ancestor of all nodes

Trees

If a path exists from node *a* to node *b*:

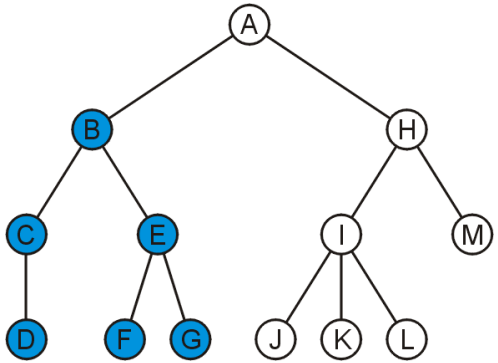
- *a* is an *ancestor* of *b*
- *b* is a *descendent* of *a*

Thus, a node is both an ancestor and a descendant of itself

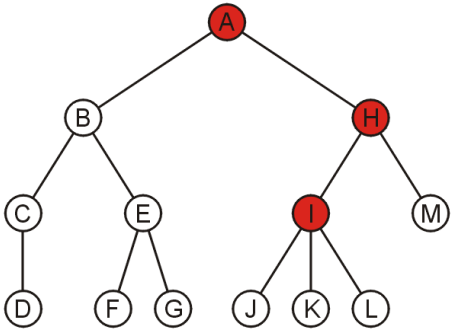
- We can add the adjective *strict* to exclude equality: *a* is a *strict descendent* of *b* if *a* is a descendant of *b* but $a \neq b$

The root node is an ancestor of all nodes

The descendants of node B are B, C, D, E, F, and G:

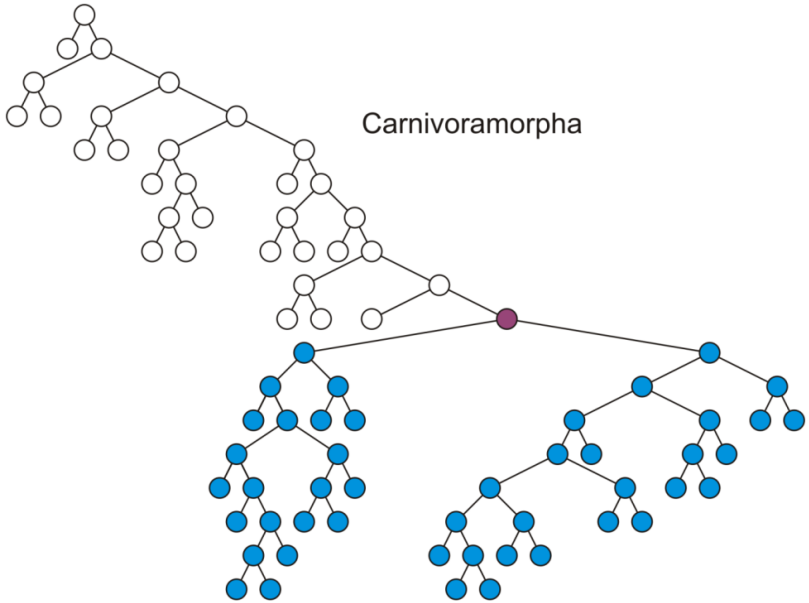


The ancestors of node I are I, H, and A:



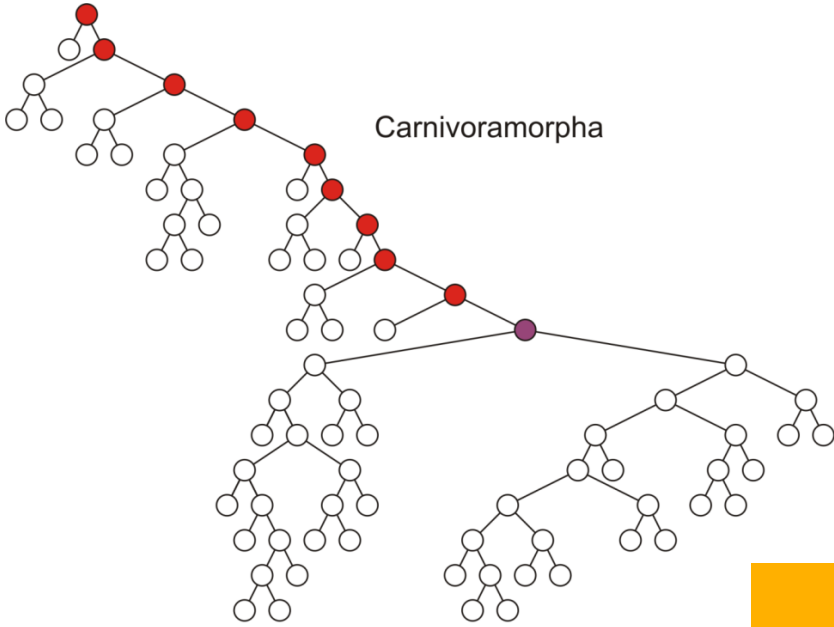
Trees

- All descendants (including itself) of the indicated node



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

All ancestors (including itself) of the indicated node



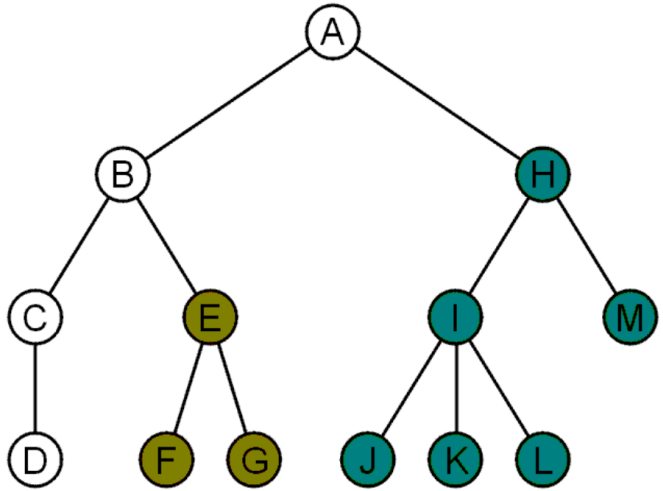
Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Trees

Another approach to a tree is to define the tree recursively:

- A degree-0 node is a tree
- A node with degree *n* is a tree if it has *n* children and all of its children are disjoint trees (*i.e.*, with no intersecting nodes)

Given any node *a* within a tree with root *r*, the collection of *a* and all of its descendants is said to be a *subtree of the tree with root a*



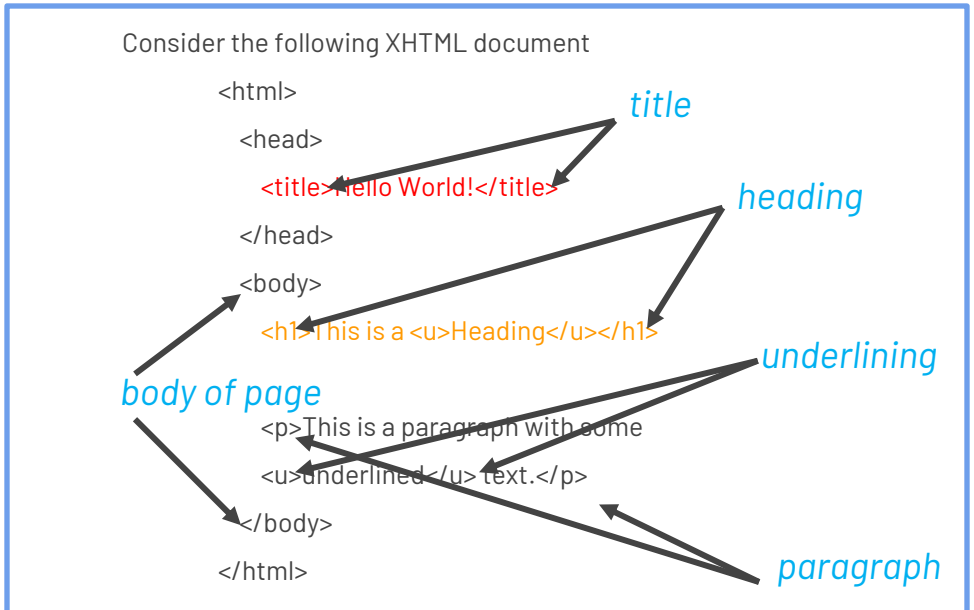
Trees

Examples

Example: XHTML

The XML of XHTML has a tree structure

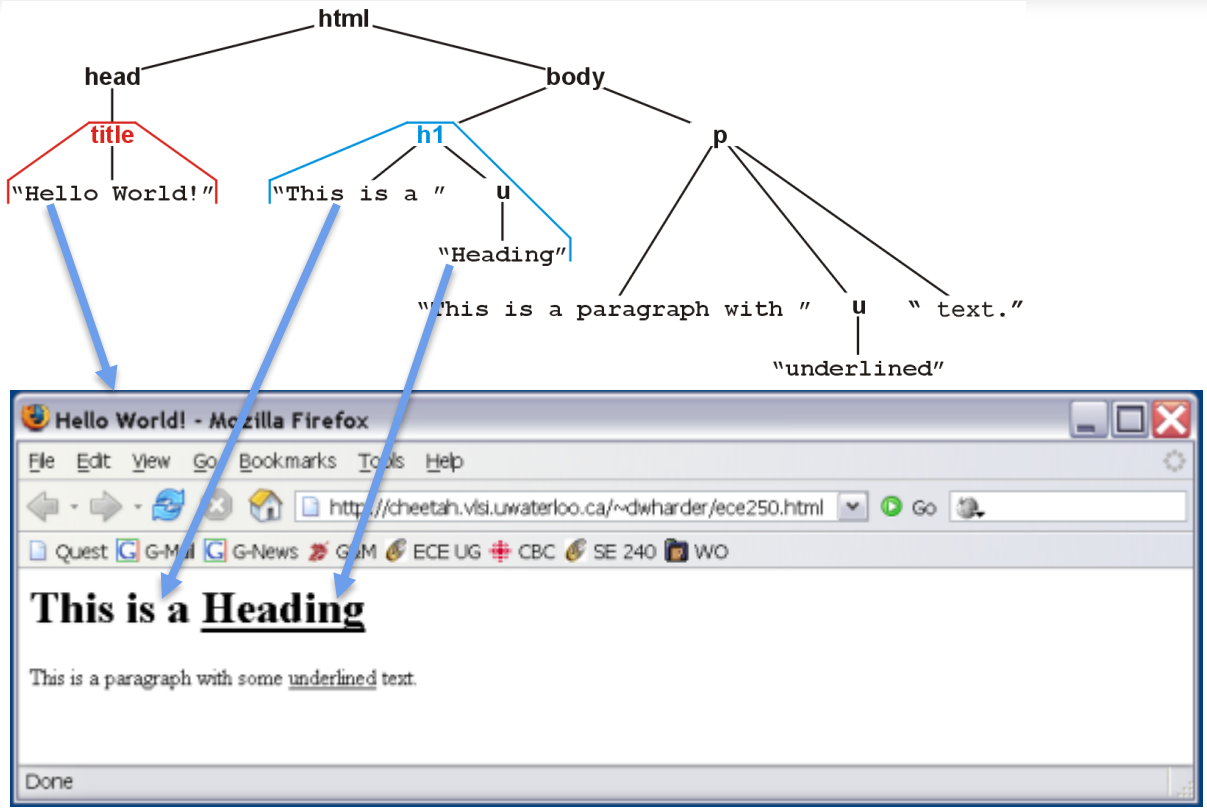
Cascading Style Sheets (CSS) use the tree structure to modify the display of HTML



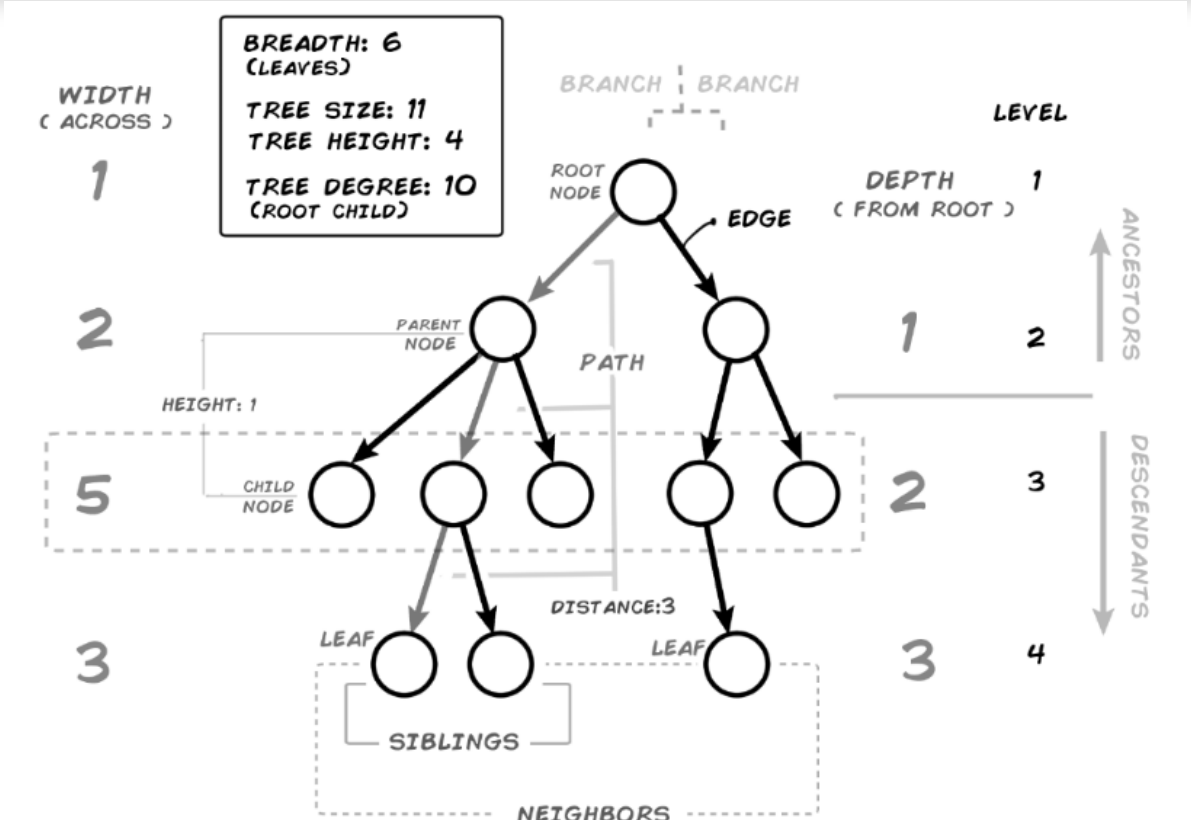
Consider the following XHTML document

```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>This is a <u>Heading</u></h1>

    <p>This is a paragraph with some
    <u>underlined</u> text.</p>
  </body>
</html>
```

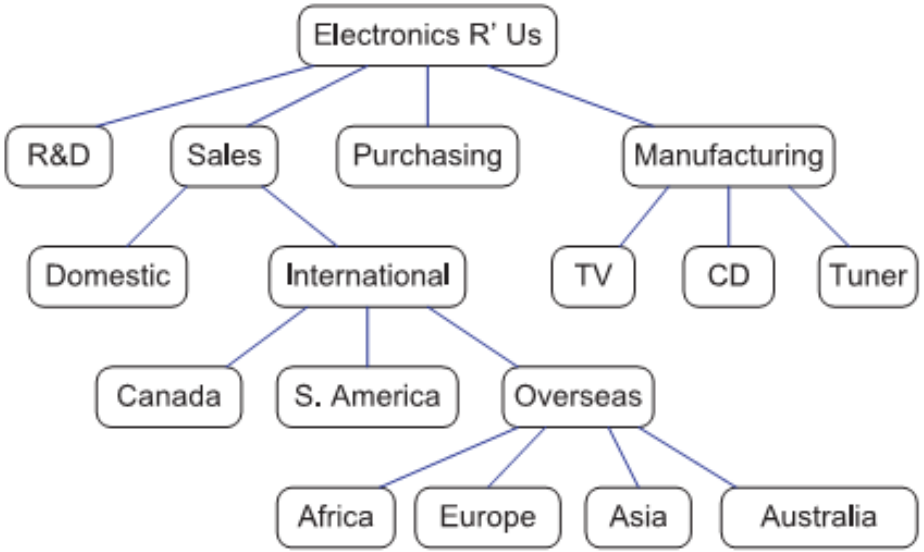
Example: XHTML

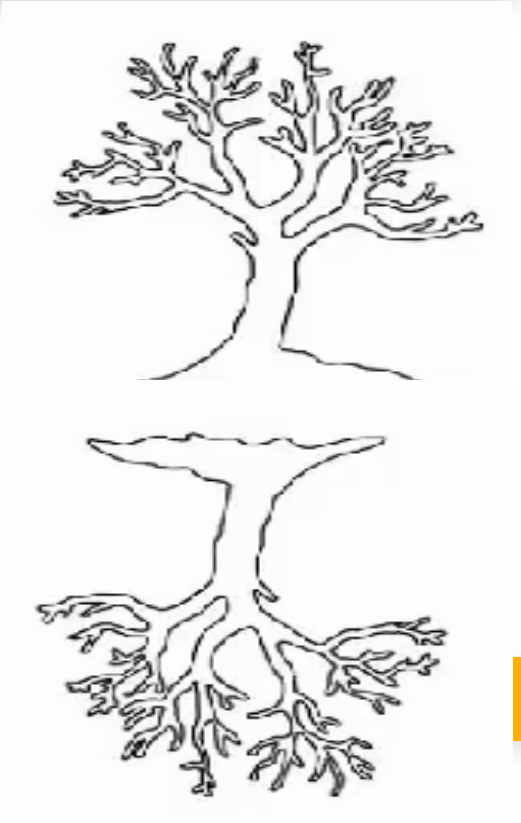


Trees


BREADTH: 6
(LEAVES)
TREE SIZE: 11
TREE HEIGHT: 4
TREE DEGREE: 10
(ROOT CHILD)
WIDTH
(ACROSS)
BRANCH
BRANCH
LEVEL
1
ROOT
NODE
EDGE
DEPTH
(FROM ROOT)
1
2
PARENT
NODE
PATH
1
2
ANCESTORS
HEIGHT: 1
5
CHILD
NODE
2
3
DESCENDANTS
DISTANCE:3
3
LEAF
LEAF
3
4
SIBLINGS
NEIGHBORS

Trees


Electronics R' Us
R&D
Sales
Purchasing
Manufacturing
Domestic
International
TV
CD
Tuner
Canada
S. America
Overseas
Africa
Europe
Asia
Australia



Trees

Traversal

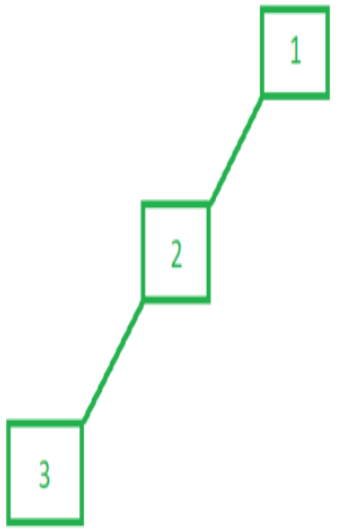
Trees

- **List of a tree's nodes** is called a traversal if it lists each tree node exactly once.
- Traversal is a process to visit all the nodes of a tree and may print their values.
- A traversal of a tree T is a systematic way of accessing, or “visiting,” all the nodes
- The three most commonly used traversal orders are recursively described as:
  - **Inorder:** traverse left subtree, visit current node, traverse right subtree
  - **Postorder:** traverse left subtree, traverse right subtree, visit current node
  - **Preorder:** visit current node, traverse left subtree, traverse right subtree

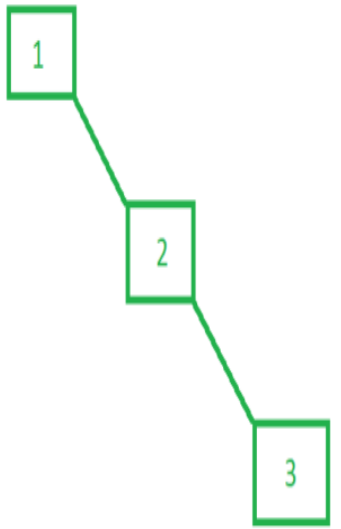
DATA STRUCTURES AND ALGORITHMS

PROBLEM WITH BST

- Typically, a binary search tree will support insertion, deletion, and search operations.
- The cost of each operation depends upon the height of the tree – in the worst case, an operation will need to traverse all the nodes on the path from the root to the deepest leaf.
- A problem starts to emerge here if our tree is heavily skewed.



LEFT SKEWED



RIGHT SKEWED

SELF BALANCING TREES

- AVL Trees
- Red Black Trees
- Splay Trees

APPLICATIONS OF SELF BALANCED TREES

- Most in-memory sets and dictionaries are stored using AVL trees.
- Database applications, where insertions and deletions are less common but frequent data lookups are necessary, also frequently employ AVL trees.
- In addition to database applications, it is employed in other applications that call for better searching.
- A balanced binary search tree called an AVL tree uses rotation to keep things balanced.
- It may be used in games with plotlines as well.
- It is mostly utilized in business sectors where it is necessary to keep records on the employees that work there and their shift changes.

AVL TREES

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as balanced binary search tree
- In AVL trees, each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

BALANCE FACTOR

- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = Height of Left Subtree - Height of Right Subtree

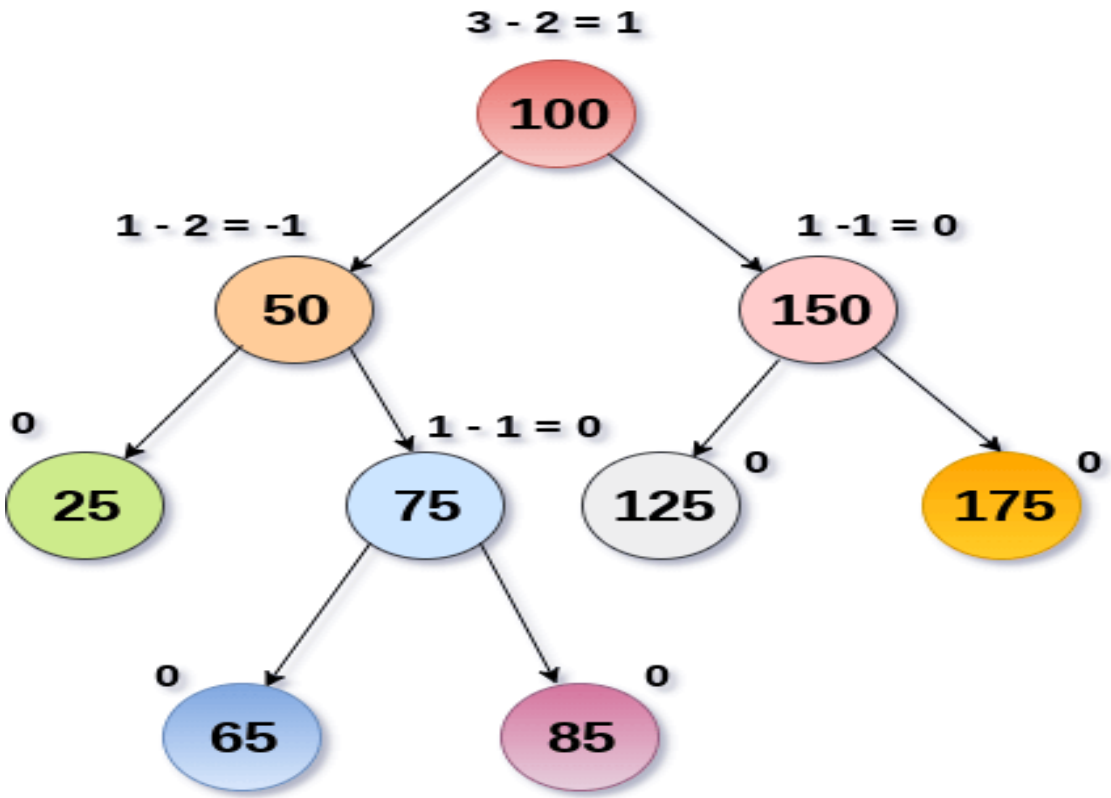
or

Balance Factor = Height of Right Subtree - Height of Left Subtree

BALANCE FACTOR

- In AVL trees, **Balance Factor** must be at most one.
- Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.
- If the value of balance factor is 0, it means that height of both subtrees are equal.
- If the value of balance factor is -1 or +1 it means that its left or right subtree height is not equal. However, the tree is in balanced condition.
- Tree is not balanced if the value exceeds +1 or -1

AVL TREE



AVL Tree

HOW TO BALANCE THE TREE?

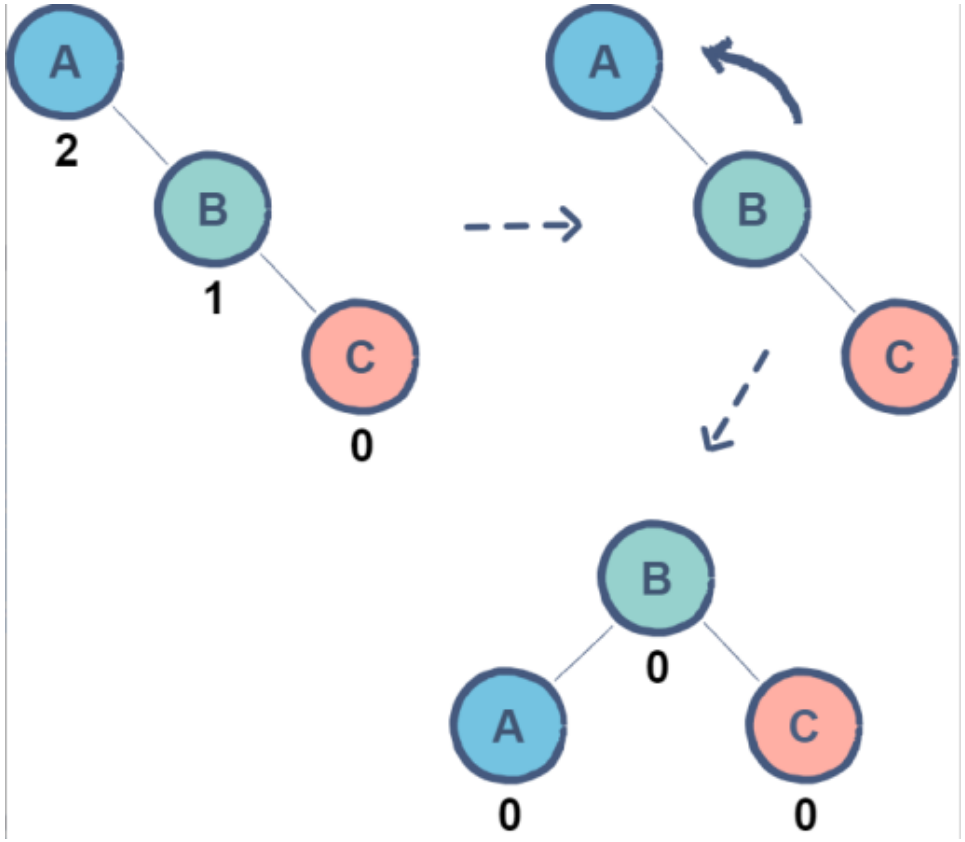
- If the balance factor is not 0, +1, -1 then the tree is needed to be balanced.
- Basically, balance factor is checked each time an item is inserted or deleted from the tree
- To make a balanced BST/ AVL tree, rotations need to be performed.
- Hence, at each insertion or deletion balance factor of the tree is checked and rotation is performed if needed

ROTATIONS IN AVL TREES

- There are following types of rotations:
  - **Single Rotation**
    - Left Rotation
    - Right Rotation
  - **Double Rotation**
    - Left Right Rotation
    - Right Left Rotation

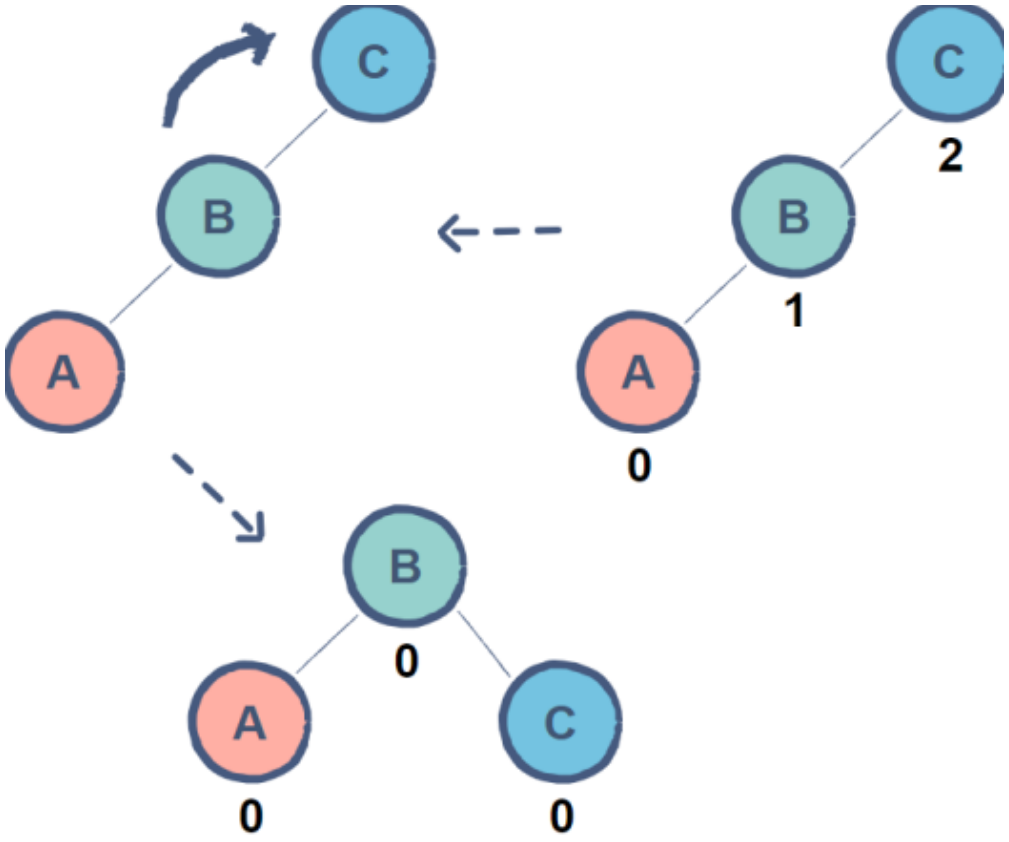
LEFT ROTATION

- When RR imbalance occurs, then left rotation is performed
- A single rotation applied when a node is inserted in the right subtree of a right subtree.
- In the given example, node A has a balance factor of 2 after the insertion of node C.
- By rotating the tree left, node B becomes the root resulting in a balanced tree.



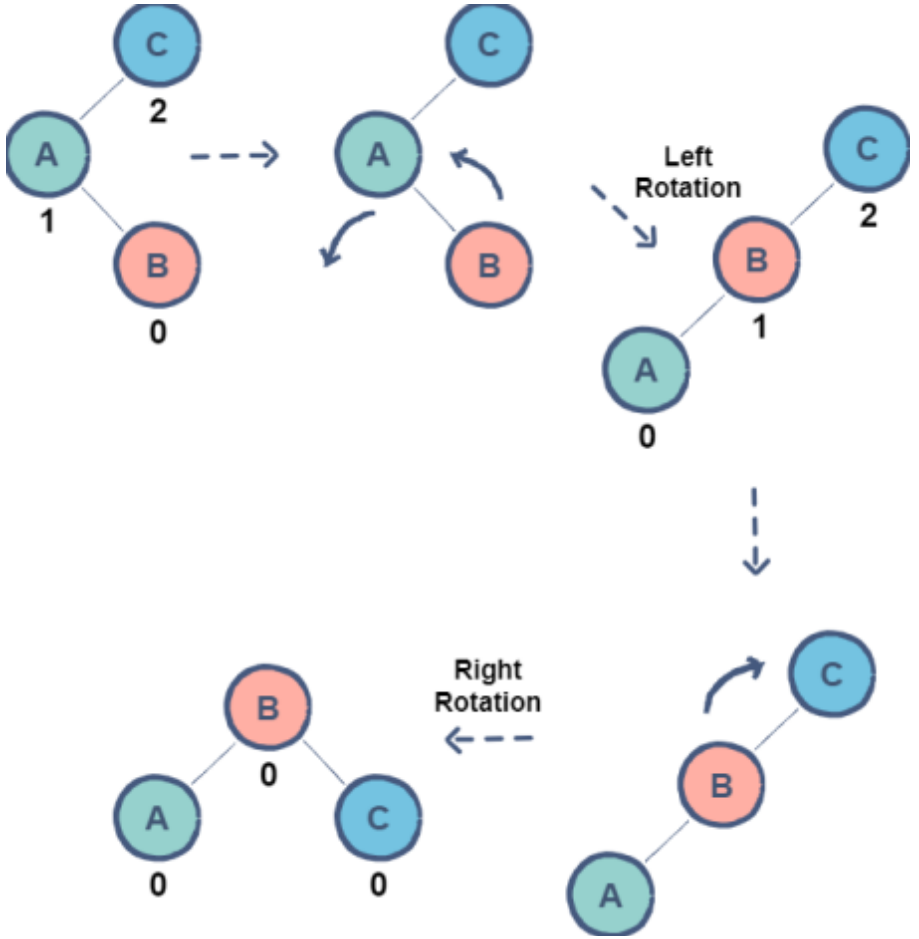
RIGHT ROTATION

- When LL imbalance occurs, then right rotation is performed
- A single rotation applied when a node is inserted in the left subtree of a left subtree.
- In the given example, node A has a balance factor of 2 after the insertion of node C.
- By rotating the tree left, node B becomes the root resulting in a balanced tree.



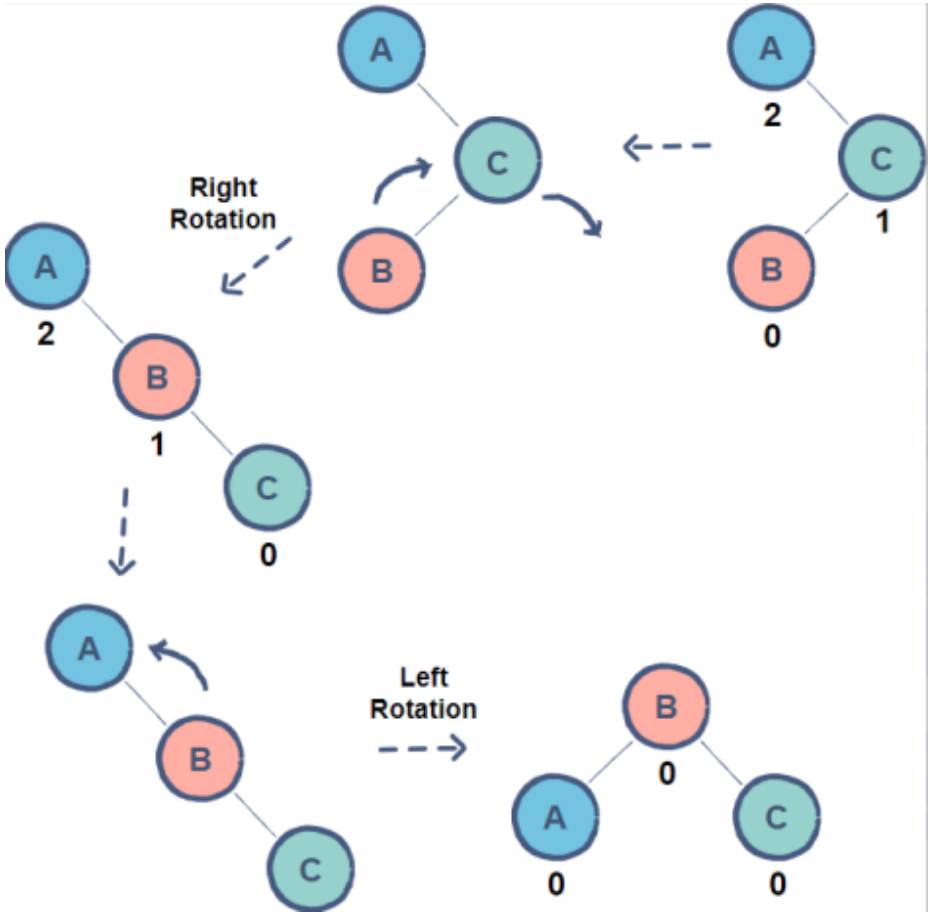
LEFT RIGHT ROTATION

- A double rotation in which a left rotation is followed by a right rotation.
- Convert the LR rotation to LL imbalance using **left rotation**
- Then solve the LL imbalance using **right rotation**
- In the given example, node B is causing an imbalance resulting in node C to have a balance factor of 2. As node B is inserted in the right subtree of node A, a left rotation needs to be applied.
- Now, all we have to do is apply the right rotation as shown before to achieve a balanced tree.



RIGHT LEFT ROTATION

- A double rotation in which a right rotation is followed by a left rotation.
- Convert the RL rotation to RR imbalance using **right rotation**
- Then solve the RR imbalance using **left rotation**
- In the given example, node B is causing an imbalance resulting in node A to have a balance factor of 2. As node B is inserted in the left subtree of node C, a right rotation needs to be applied.
- Now, by applying the left rotation as shown before, we can achieve a balanced tree.



DATA STRUCTURES AND ALGORITHMS

INSERTION IN BST

- Check the value to be inserted (say **X**) with the value of the current node (say **val**) we are in:
  - If **X** is less than **val** move to the left subtree.
  - Otherwise, move to the right subtree.
- Once the leaf node is reached, insert **X** to its right or left based on the relation between **X** and the leaf node's value.

INSERTION IN BST

```
void BST::insert(int value)
{
	//Node create:
	Node*nn = new Node;
	nn->data = value;
	nn->leftChild = nullptr;
	nn->rightChild = nullptr;

	if (root == nullptr) //empty tree case
		root = nn;

	else { -------------- }
```

TREE TRAVERSAL

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree −
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal

IN-ORDER TRAVERSAL

- In this traversal method, traversal method is as follows:
  - the left subtree is visited first,
  - then the root
  - then the right sub-tree.
- We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

PRE-ORDER TRAVERSAL

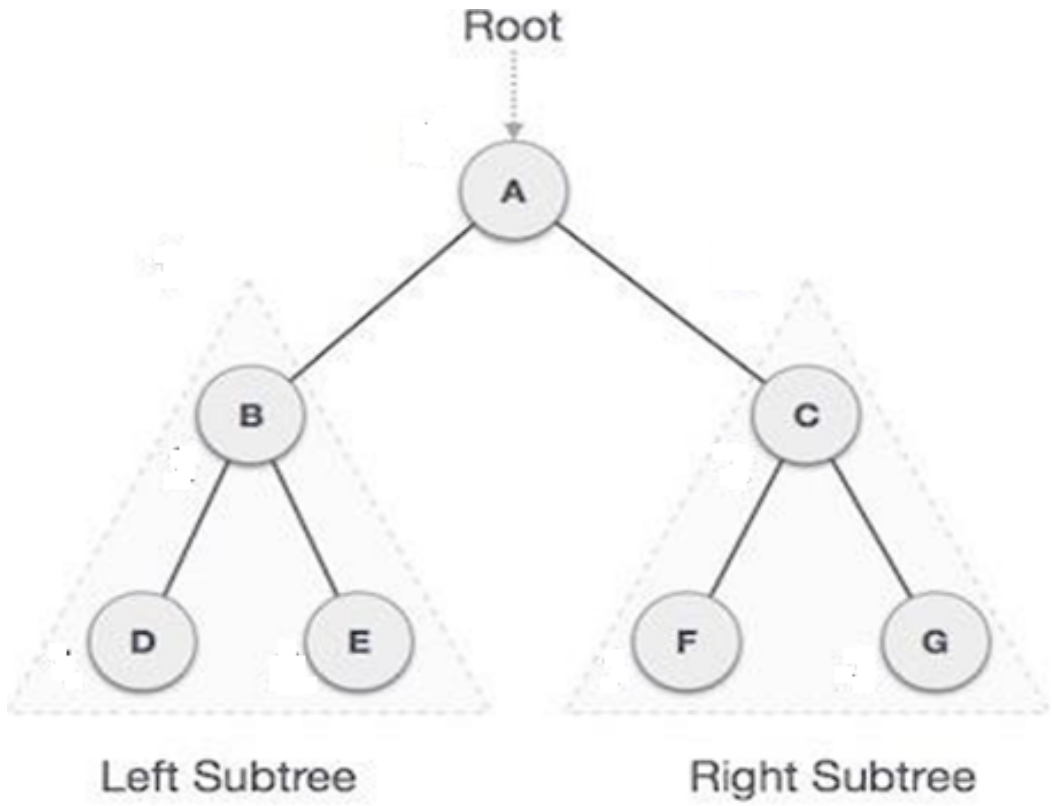
The steps to perform the preorder traversal are listed as follows -

- First, visit the root node.
- Then, visit the left subtree.
- At last, visit the right subtree.

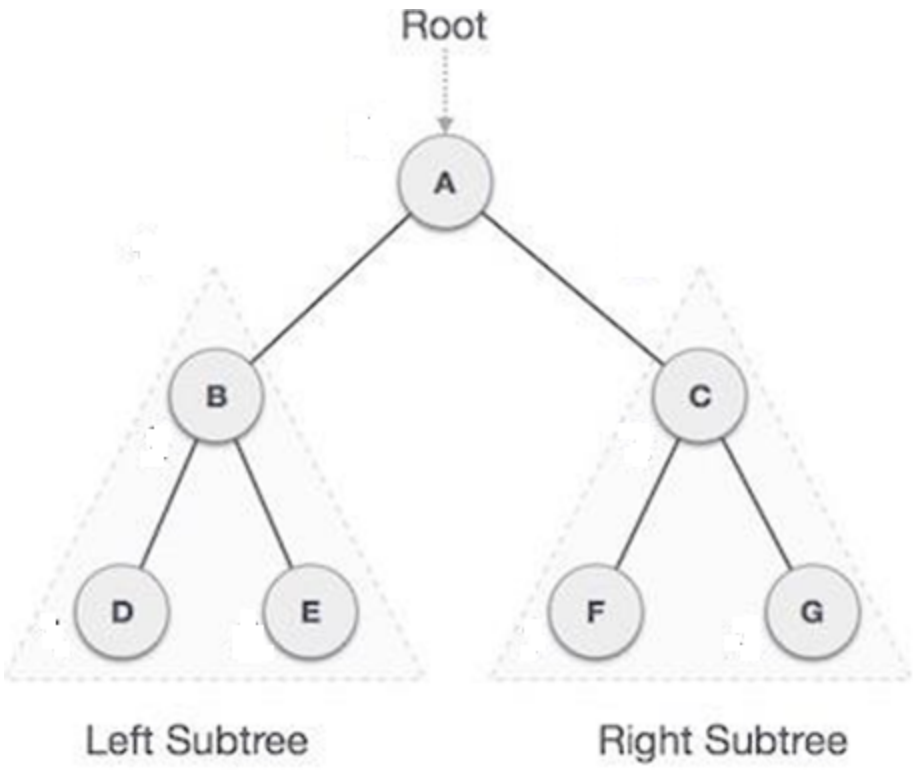
POST-ORDER TRAVERSAL

- Post-order traversal is used to get the postfix expression of a tree.
- The following steps are used to perform the post-order traversal:
- Traverse the left subtree
- Traverse the right subtree
- Root node is traversed at last

TREE TRAVERSAL

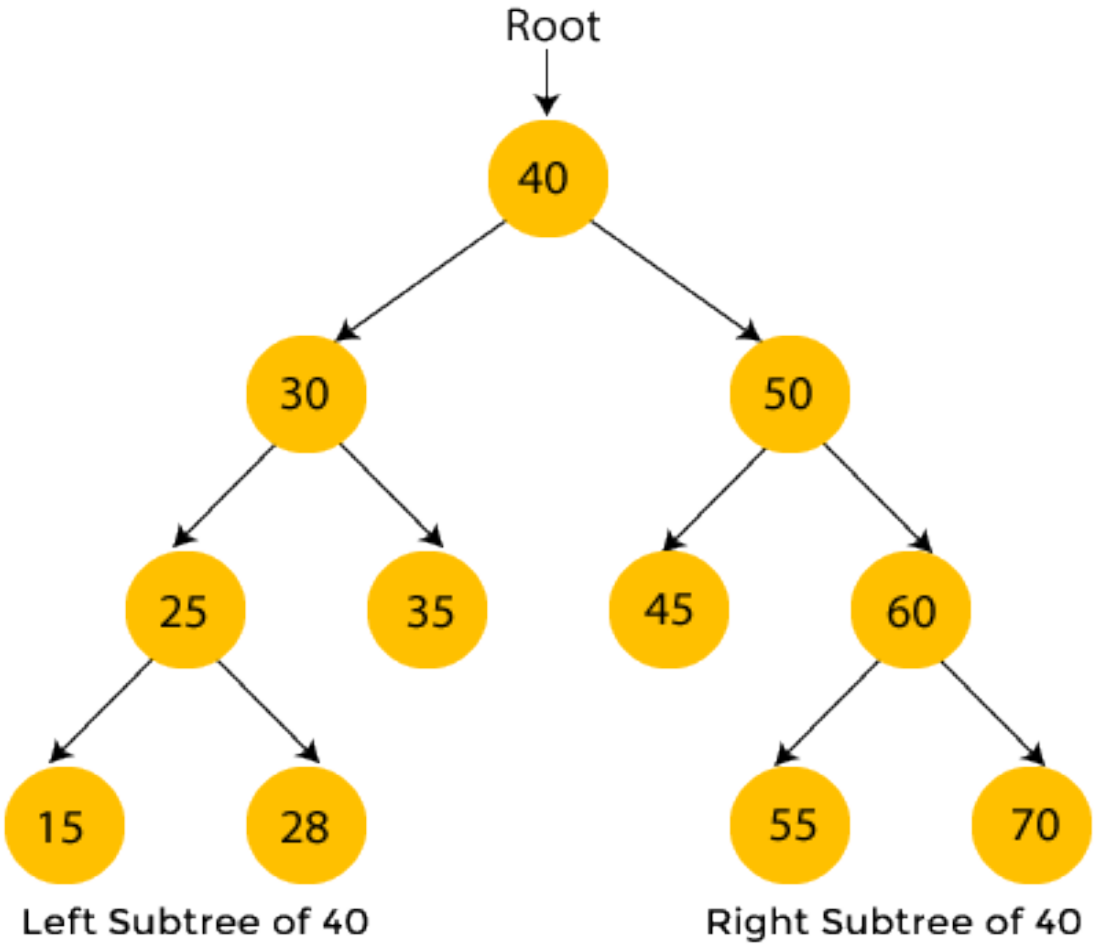


TREE TRAVERSAL

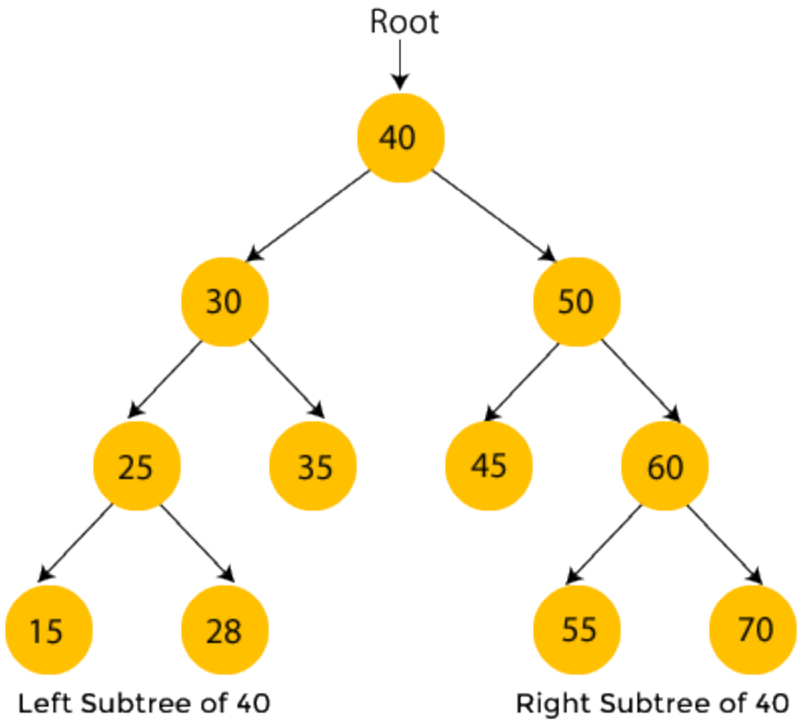


- Post-order: D → E → B → F → G → C → A
- Pre-order: A → B → D → E → C → F → G
- In-order: D → B → E → A → F → C → G

TREE TRAVERSAL



TREE TRAVERSAL

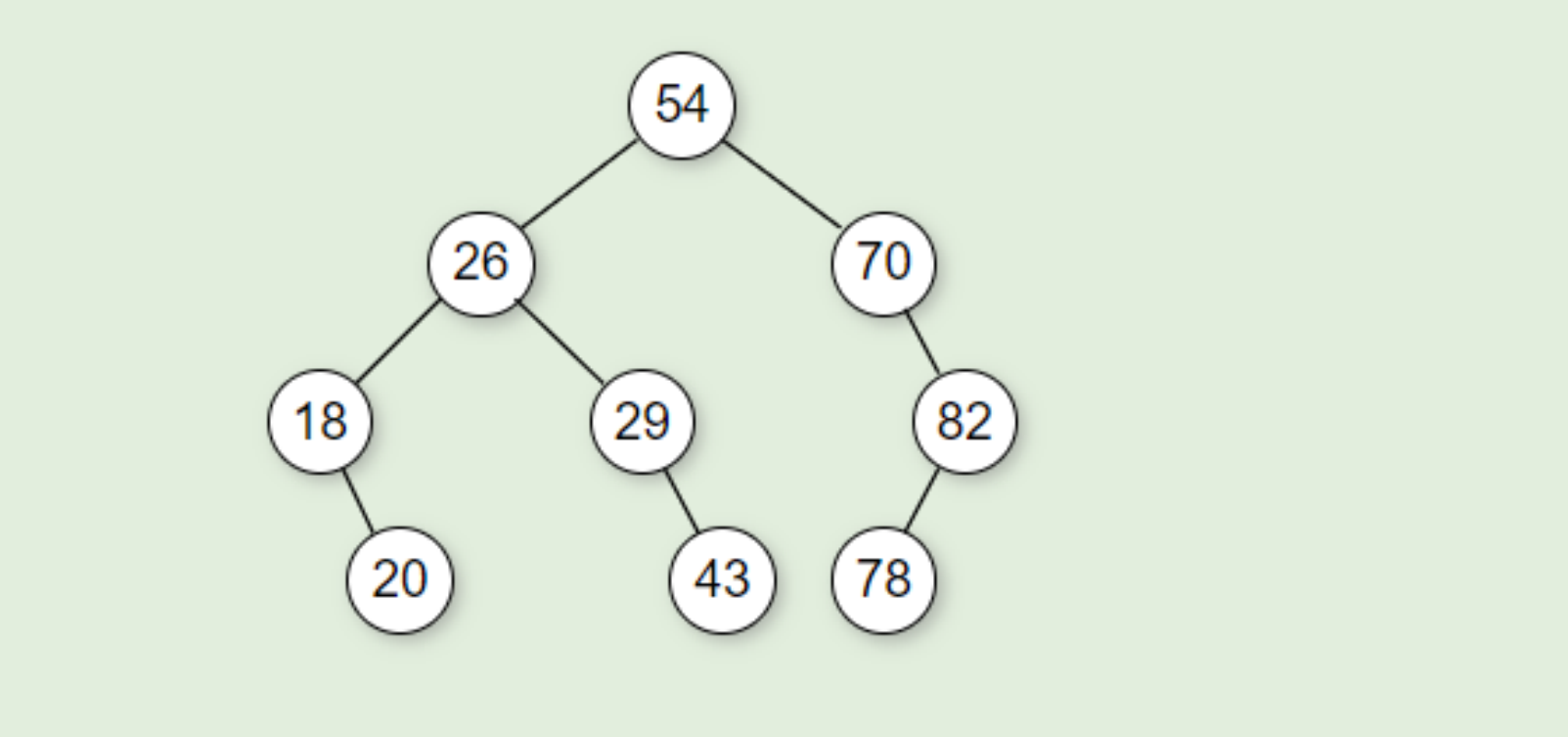


- Pre-order: 40, 30, 25, 15, 28, 35, 50, 45, 60, 55, 70
- Post-order: 15, 28, 25, 35, 30, 45, 55, 70, 60, 50, 40
- In-order: 15, 25, 28, 30, 35, 40, 45, 50, 55, 60, 70

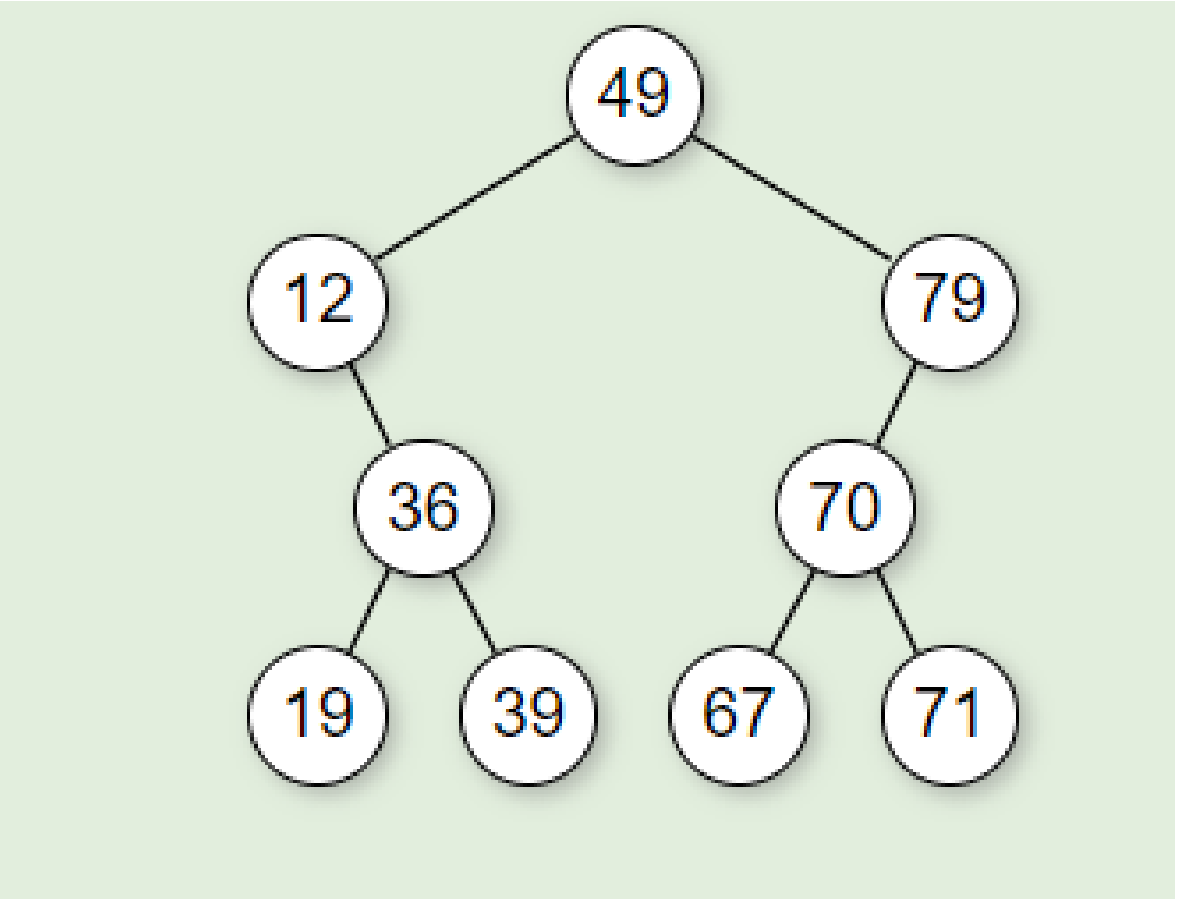
IN-ORDER TRVERSAL

```
void BST::INORDER(Node*p)
{
	if (p != nullptr)
	{
		INORDER(p->leftChild);
		cout << p->data << endl;
		INORDER(p->rightChild);
	}
}
```

TREE TRAVERSAL



TREE TRAVERSAL

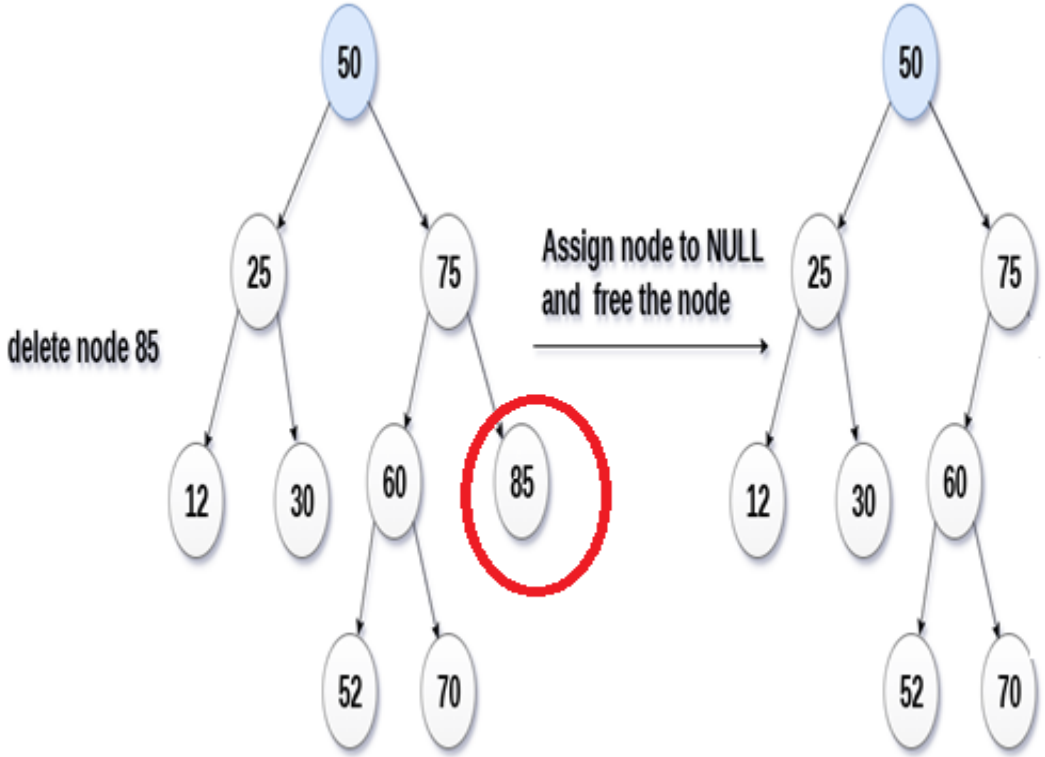


DELETION IN BST

- Delete function is used to delete the specified node from a binary search tree.
- However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.
- There are three situations of deleting a node from binary search tree.
  - The node to be deleted is a leaf node
  - The node to be deleted has only one child
  - The node to be deleted has two children.

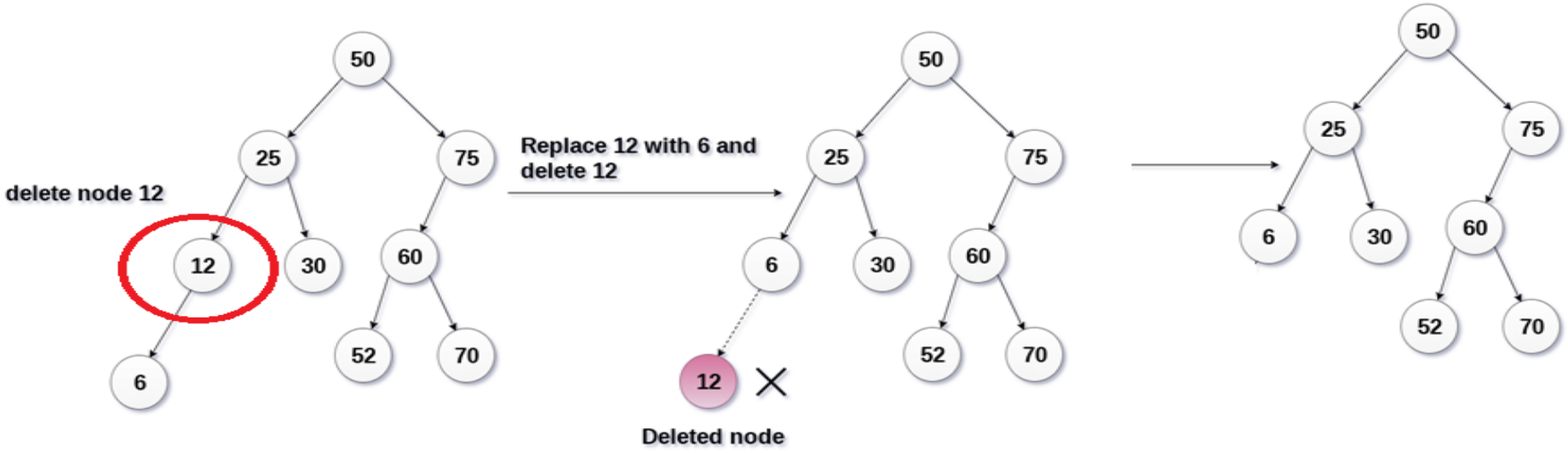
NODE TO BE DELETED IS A LEAF NODE

- It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.
- In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



NODE TO BE DELETED HAS ONE CHILD

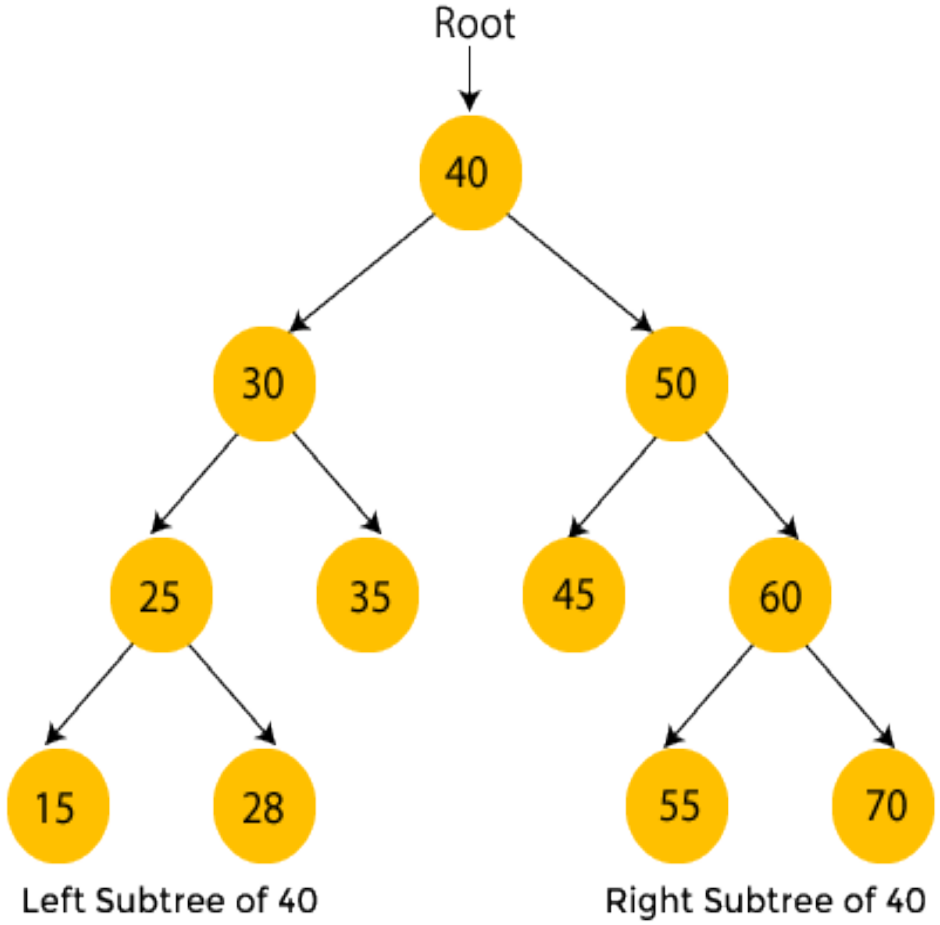
- In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.
- In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



NODE TO BE DELETED HAS TWO CHILDREN

- It is a bit complexed case compare to other two cases.
- However, the node which is to be deleted, is replaced with its in-order successor or predecessor i.e. find the smallest node in the right subtree.
- Replace the value of current node with the in-order successor or predecessor
- After the procedure, replace the node with NULL and free the allocated space.

NODE TO BE DELETED HAS TWO CHILDREN



DATA STRUCTURES AND ALGORITHMS

HASHING

- A technique to convert a range of key values into a range of indexes of an array
- When it comes to data structures, hashing is a technique used to store and retrieve data in a database
- Hashing is an efficient way to store and retrieve data in O(1), because it avoids the need for comparisons between elements
- It also allows duplicate values to be stored in the same structure without causing collisions
- It is also called the mapping technique as larger values are mapped into smaller ones
- There are many different hashing algorithms, but they all share the same basic principle

BASIC TERMINOLOGIES

- Search Keys
- Hash Tables
- Hash Functions

HASH TABLES

- The hash table is an associative data structure that stores data as associated keys
- An array format is used to store hash table data, where each value is assigned its unique index
- By knowing the index of the desired data, we can access it very quickly
- As a result, inserting and searching data is very fast, regardless of data size
- In a Hash Table, elements are stored in an array, and an index is generated using hashing techniques in a data structure

HASH TABLES

- Hash table is basically an array
- However, in hash table whenever data is inserted, deleted or searched it does not require scanning the whole data
- Hash table insert, delete and search a value with the help of hash function
- The Hash table data structure stores elements in key-value pairs where
  - Key- unique integer that is used for indexing the values
  - Value - data that are associated with keys.

HASH FUNCTION

Several hash functions are described in the literature. Here we describe some of the commonly used hash functions:

- **Mid-Square:** In this method, the hash function, h, is computed by squaring the identifier, and then using the appropriate number of bits from the middle of the square to obtain the bucket address. Because the middle bits of a square usually depend on all the characters, it is expected that different keys will yield different hash addresses with high probability, even if some of the characters are the same.
- **Folding:** In folding, the key X is partitioned into parts such that all the parts, except possibly the last parts, are of equal length. The parts are then added, in some convenient way, to obtain the hash address.
- **Division (Modular arithmetic):** In this method, the key X is converted into an integer. This integer is then divided by the size of the hash table to get the remainder, giving the address of X in HT.

HASH FUNCTION

- Value to be inserted:

  $8, 2, 10, 0, 11,$

- Calculate Hash Function/ hash value

  $h(x) = x$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		2						8		10	11		

HASH FUNCTION

- Value to be inserted:

  8, 2, 10, 0, 11, 25

- Calculate Hash Function/ hash value

  $h(x) = x$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		2						8		10	11		

HASH FUNCTION (DIVISION METHOD)

- Value to be inserted:

  $24, 52, 91, 67, 48, 83$

- Calculate Hash Function/ hash value

  $h(x) = x \ \% \ size\ of\ table$

0	1	2	3	4	5	6	7	8	9
	91	52	83	24			67	48	

HASH FUNCTION (DIVISION METHOD)

- Value to be inserted:

  $8, 66, 9, 57, 20,$

- Calculate Hash Function/ hash value

  $h(x) = x \ \% \ size\ of\ table$

0	1	2	3	4	5	6	7	8	9
20						66	57	8	9

HASH FUNCTION (DIVISION METHOD)

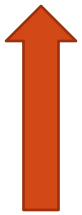
- Value to be inserted:

  $8, 66, 9, 57, 20, 43, 76$

- Calculate Hash Function/ hash value

  $h(x) = x \ \% \ size\ of\ table$

0	1	2	3	4	5	6	7	8	9
20			3			66	57	8	9



There is already some data at index 6

This is called collision

COLLISION

- A hash function gets us a small number for a key which is a big integer or string
- There is a possibility that two keys result in the same value
- The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision
- It must be handled using some collision handling technique

COLLISION RESOLUTION TECHNIQUES

- Open Hashing/ Closed Addressing
  - Separate Chaining

- Closed Hashing/ Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

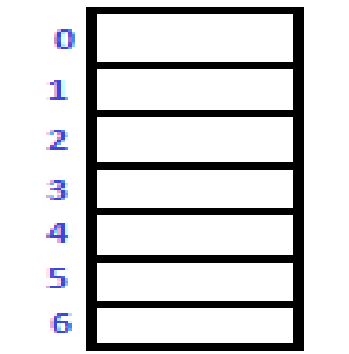
SEPARATE CHAINING

- The idea behind separate chaining is to implement the array as a linked list called a chain
- Separate chaining is one of the most popular and commonly used techniques in order to handle collisions
- So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain

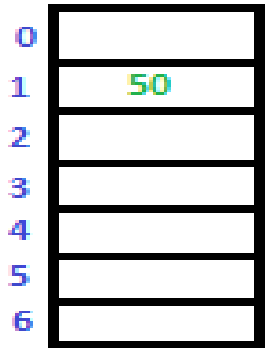
SEPARATE CHAINING

- Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101

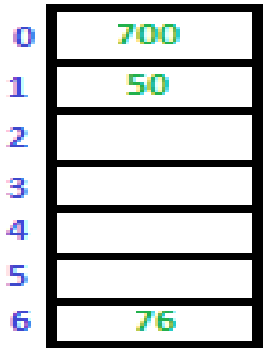
SEPARATE CHAINING



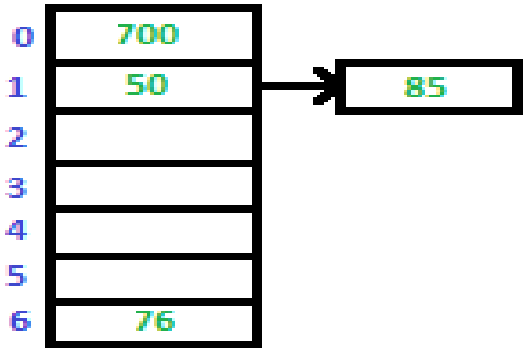
Initial Empty Table



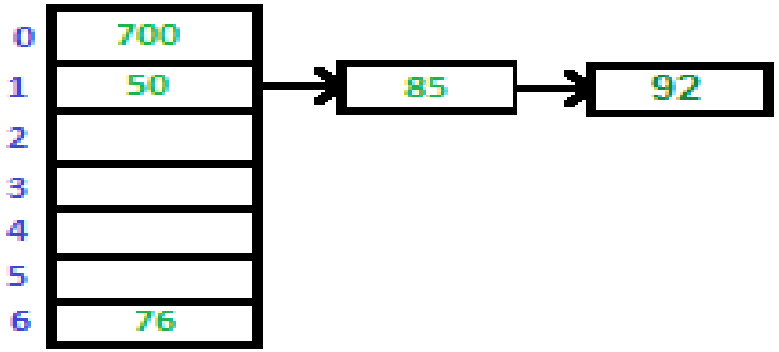
Insert 50



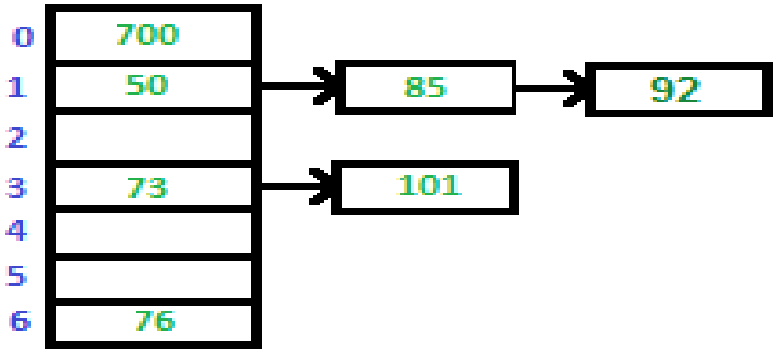
Insert 700 and 76



Insert 85: Collision Occurs, add to chain



Inser 92 Collision Occurs, add to chain



Insert 73 and 101

SEPARATE CHAINING

Advantages

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages

- The cache performance of chaining is not good as keys are stored using a linked list.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

OPEN ADDRESSING/ CLOSED HASHING

- Linear Probing
- Quadratic Probing
- Double Hashing

LINEAR PROBING

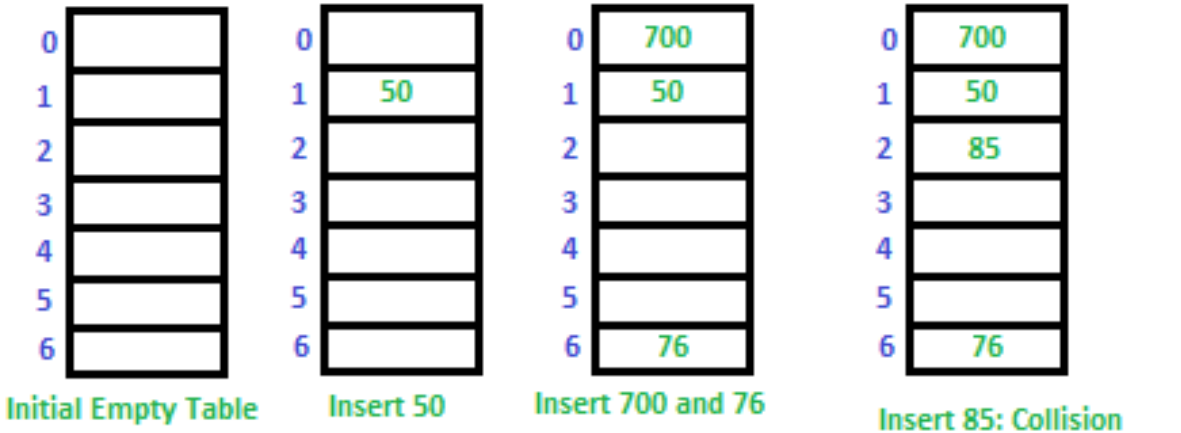
- In linear probing, the hash table is searched sequentially that starts from the original location of the hash.
- **Calculate Hash Function/ hash value**

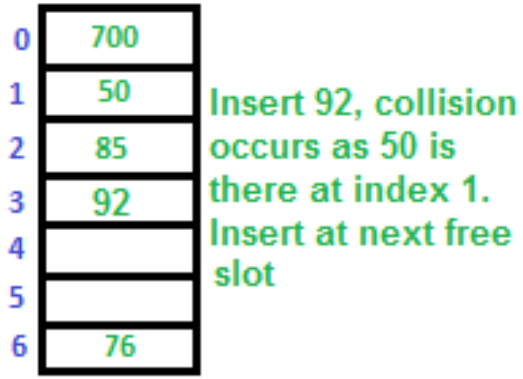
$$h(x) = x \,\%\, size\ of\ table$$

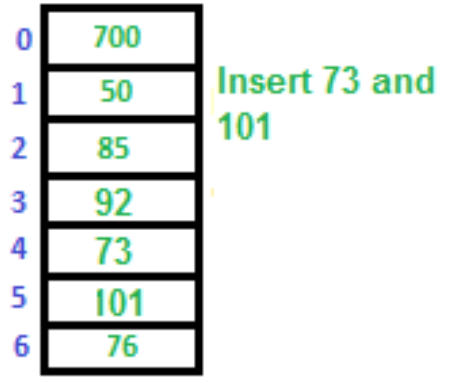
- If in case the location that we get is already occupied, then we check for the next location.
- **Calculate Hash Function/ hash value again**

$$h'(x) = (h(x) + i) \,\%\, size\ of\ table$$

LINEAR PROBING







LINEAR PROBING

Advantages

- Overall, linear probing is a simple and efficient method for handling collisions in hash tables
- It can be used in a variety of applications that require efficient storage and retrieval of data
- No extra space is wasted

Disadvantages

- Searching time is O(n)
- Two main challenges for linear probing are:
  - **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
  - **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

QUADRATIC PROBING

- If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value.
- Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed before.
- This method is also known as the **mid-square** method.
- In this method, we look for the i^2th slot in the ith iteration.
- We always start from the original hash location.
- If only the location is occupied then we check the other slots.

QUADRATIC PROBING

- In quadratic probing, the hash table is searched sequentially that starts from the original location of the hash.
- **Calculate Hash Function/ hash value**

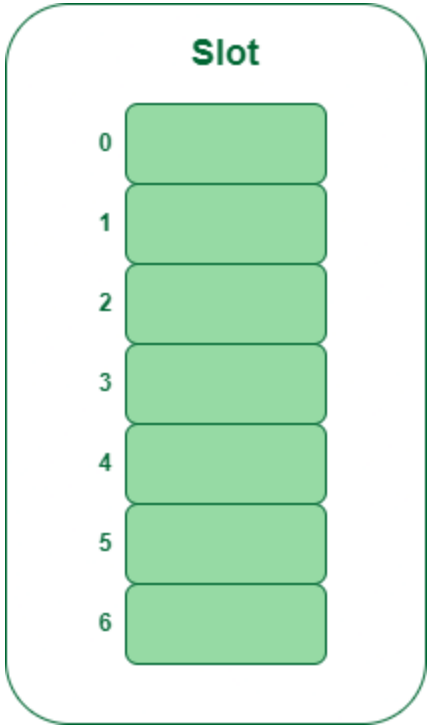
$$h(x) = x \,\%\, size\ of\ table$$

- If in case the location that we get is already occupied, then we check for the next location.
- **Calculate Hash Function/ hash value again**

$$h'(x) = (h(x) + i^2) \,\%\, size\ of\ table$$

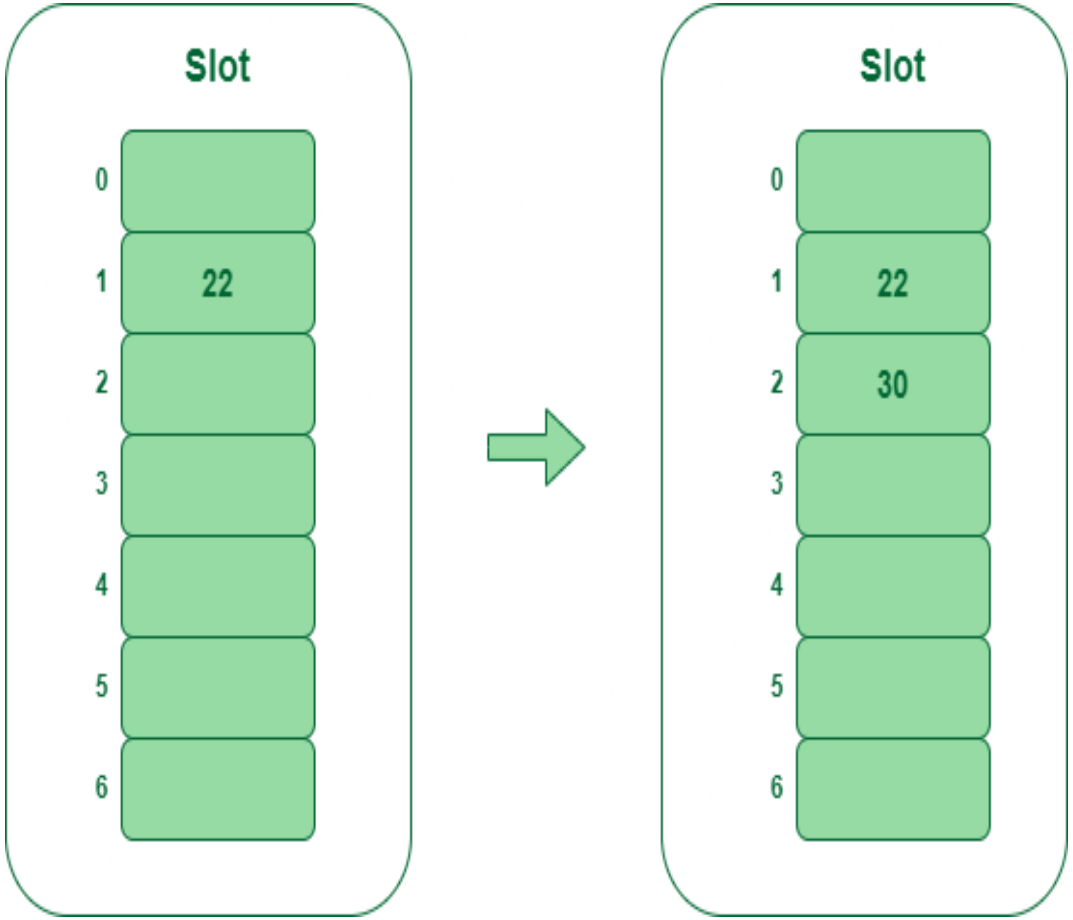
QUADRATIC PROBING

- Let us consider table Size = 7, hash function as Hash(x) = x % 7
- Collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50.
- **Step 1:** Create a table of size 7.



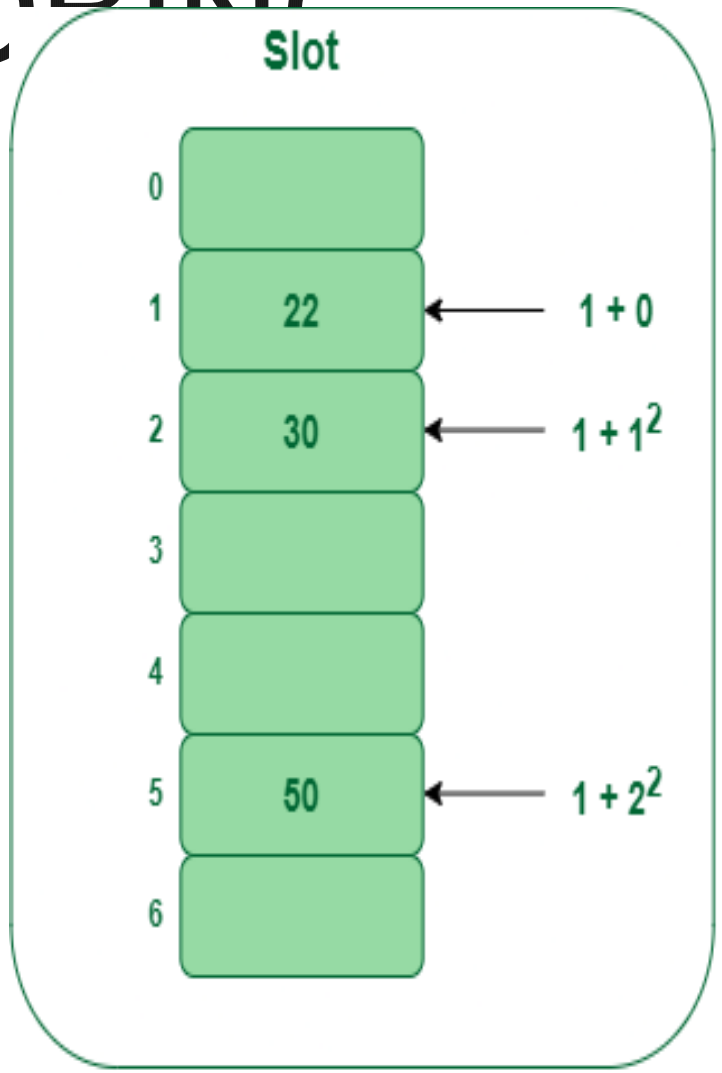
QUADRATIC PROBING

- Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
- Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



QUADRATIC PROBING

- Hash(50) = 50 % 7 = 1
- In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
- Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
- Now, cell 5 is not occupied so we will place 50 in slot 5.



QUADRATIC PROBING

Advantages

- No extra space is wasted
- Primary clustering issue is resolved

Disadvantages

- Secondary clustering issue
- Search time is O(n)
- No guarantee of finding any slot

DOUBLE HASHING

- The intervals that lie between probes are computed by another hash function.
- Double hashing is a technique that reduces clustering in an optimized way.
- In this technique, the increments for the probing sequence are computed by using another hash function.
- We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i^{th}** rotation.

DOUBLE HASHING

- **Calculate Hash Function/ hash value**

$$h1(x) = x \,\%\, size\ of\ table$$

- If in case the location that we get is already occupied, then we check for the next location.
- **Calculate Hash Function/ hash value again**

$$h'(x) = (h1(x) + i * h2(x)) \,\%\, size\ of\ table$$

where,

DOUBLE HASHING

- If size of table is any prime number,

$$h2(x) = 1 + (x \ mod \ (size \ of \ table - 2))$$

- If size of table is not prime number, then you must select the prime number smaller than the size of table.

$$h2(x) = R \ - x \ mod \ R$$

where R is any prime number smaller than the size of table.

- For example, if the size of table is 13 then

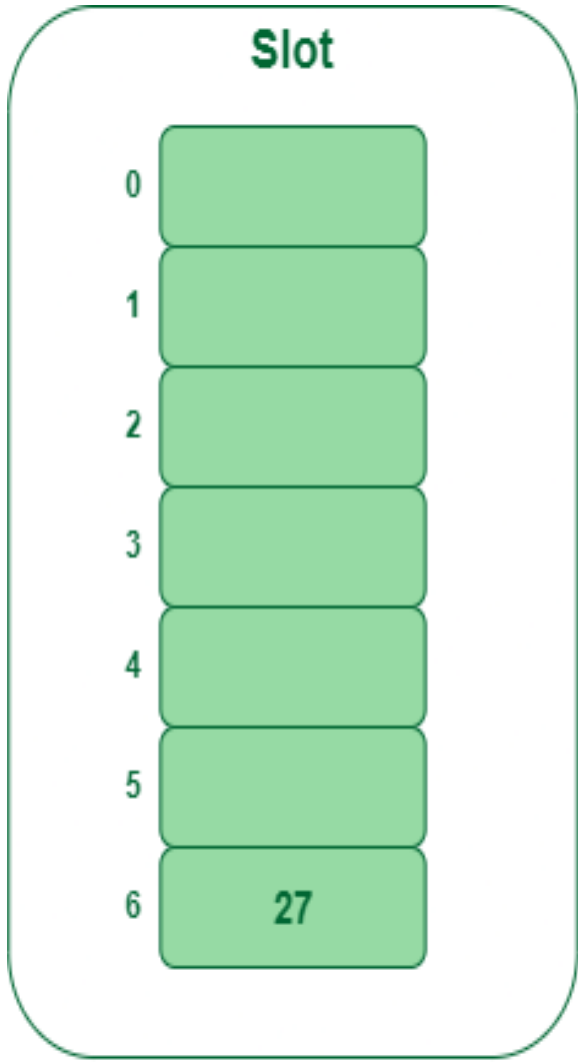
$$h2(x) = 1 + (x \ mod \ (13 - 2))$$
$$h2(x) = 1 + (x \ mod \ (11))$$

- For example, if the size of table is 10 then

$$h2(x) = 7 \ - x \ mod \ 7$$

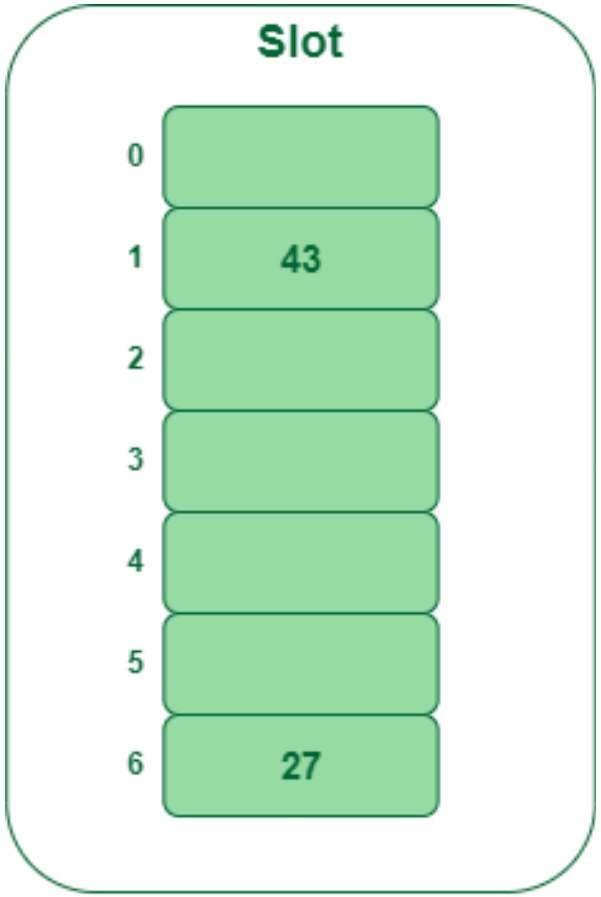
DOUBLE HASHING

- Insert the keys 27, 43, 692, 72 into the Hash Table of size 7.
- First hash-function is h1(x) = x mod 7
- Second hash-function is h2(x) = 1 + (x mod 5)
- Insert 27



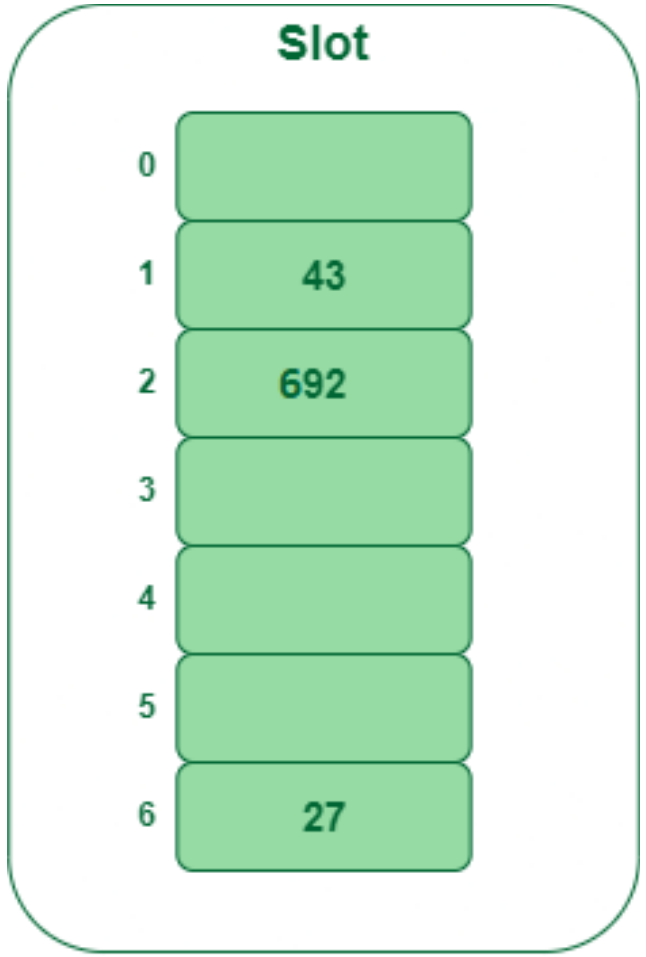
DOUBLE HASHING

- Insert 43 as the desired location is empty



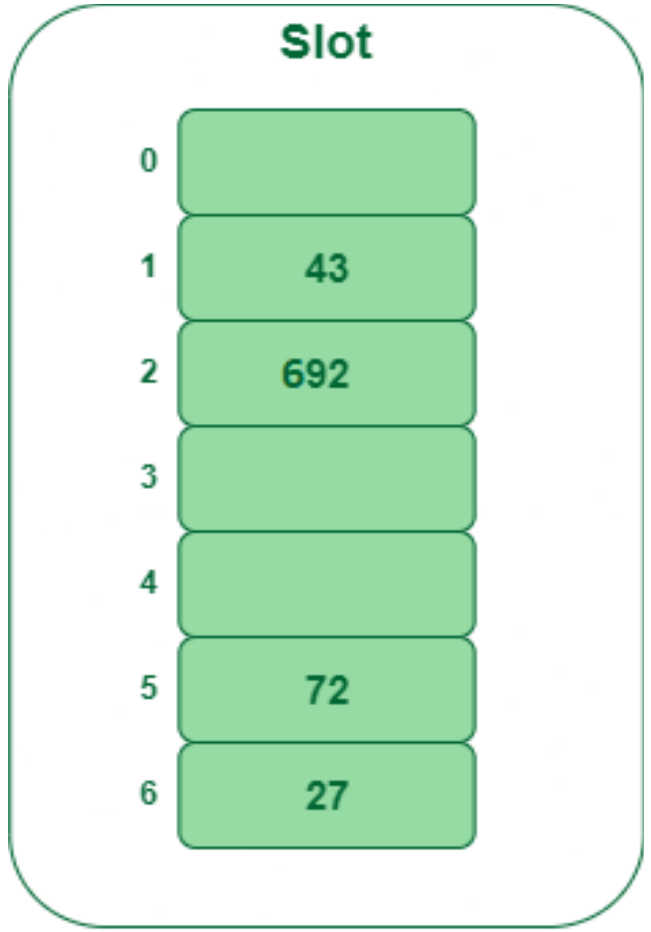
DOUBLE HASHING

- Insert 692
- 692 % 7 = 6, but location 6 is already being occupied and this is a collision
- So we need to resolve this collision using double hashing.



DOUBLE HASHING

- Insert 72
- 72 % 7 = 2, but location 2 is already being occupied and this is a collision
- So we need to resolve this collision using double hashing.



DOUBLE HASHING

Advantages

- No extra space is wasted
- No primary clustering
- No secondary clustering
- Double hashing is useful if an application requires a smaller hash table since it effectively finds a free slot.
- It produces a uniform distribution of records throughout a hash table.

Disadvantages

- O(n) time complexity

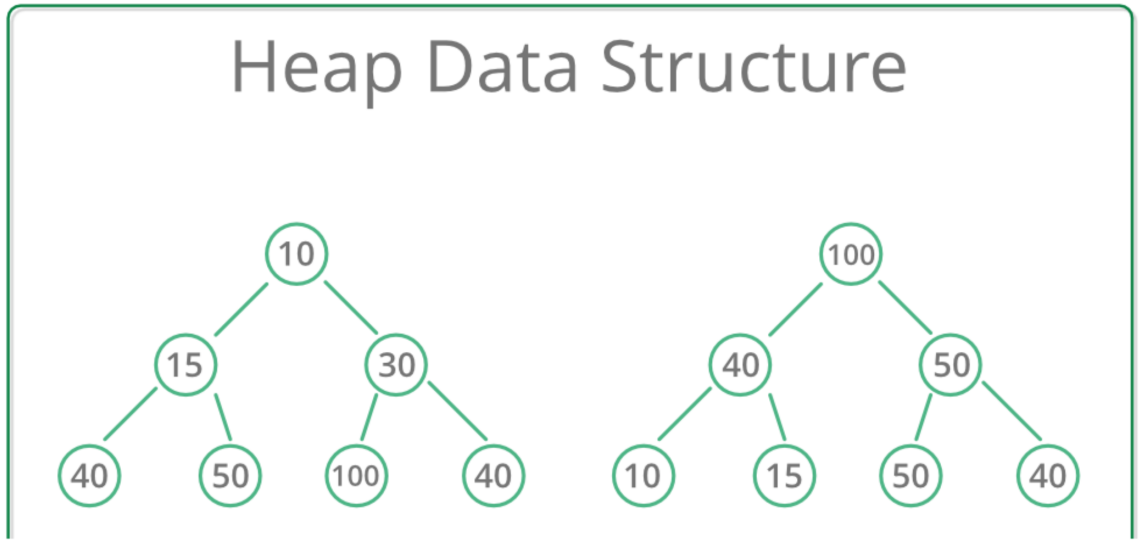
DOUBLE HASHING: EXAMPLE

- Suppose there are six students in the Data Structures class and their IDs are 115, 153, 586, 206, 985, and 111, respectively. We want to store each student's data in this order. Suppose HT is of the size 19.
- How will the data is stored in HT using Double Hashing

DATA STRUCTURES AND ALGORITHMS

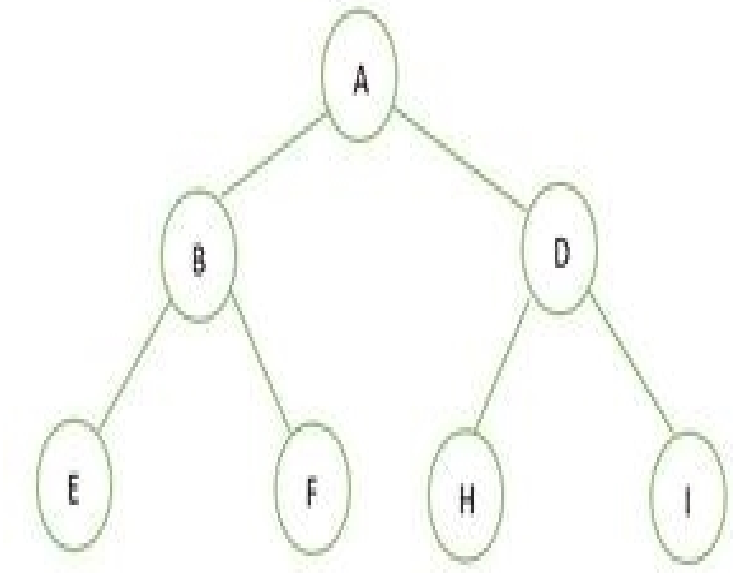
HEAP

- Heap is a tree based data structure
- It follows the following properties:
  - Structural Property
  - Ordering Property

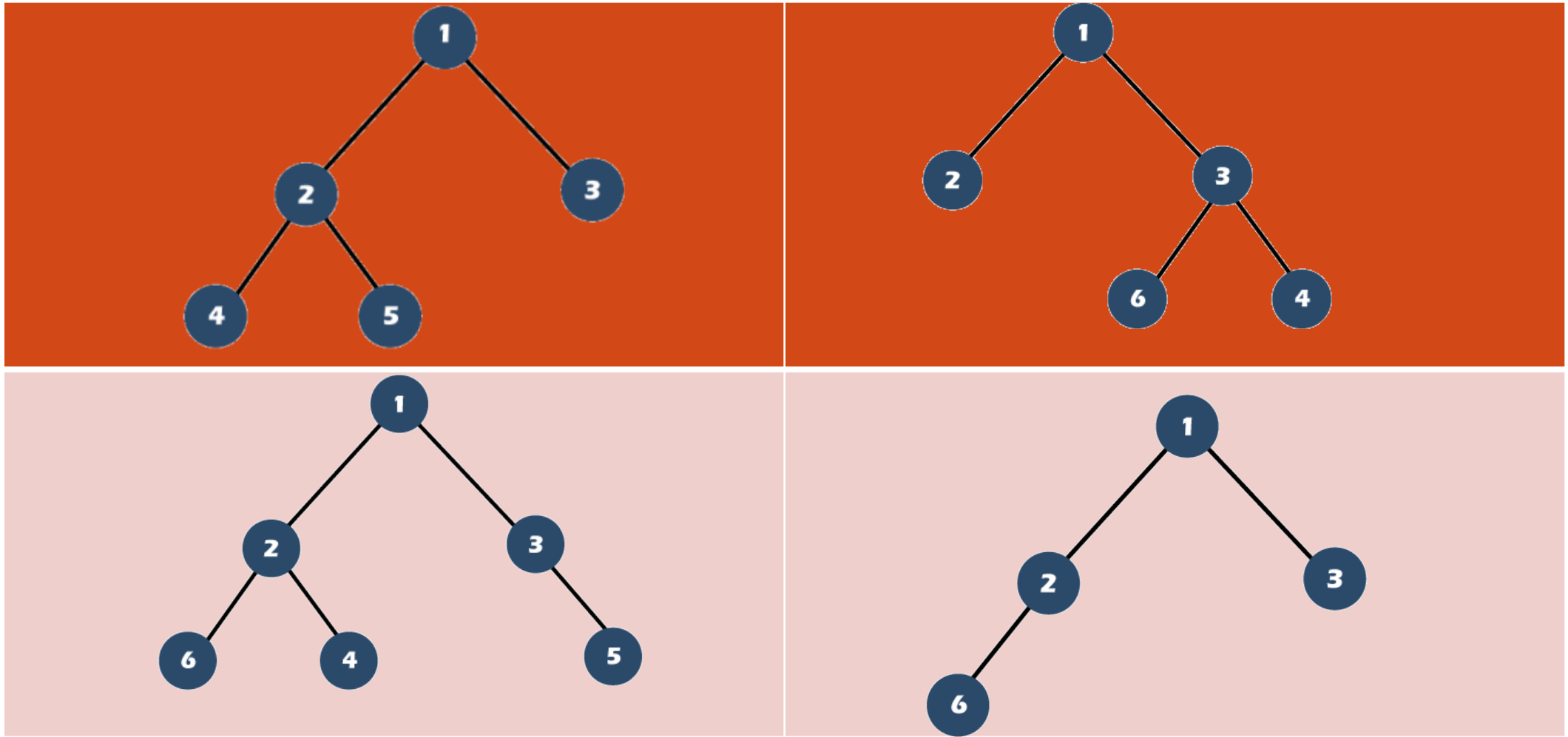


STRUCTURAL PROPERTY OF HEAP

- Structural property of heap says that the tree must be a complete binary tree.
- A complete binary tree is a binary tree in which all the levels of binary tree is completely filled except possibly the last level, which is filled from left side.
- All the levels should be filled except for the last one. The last level may or may not be filled.
- All the nodes should be filled from left to right.



COMPLETE BINARY TREE



ORDERING PROPERTY OF HEAP

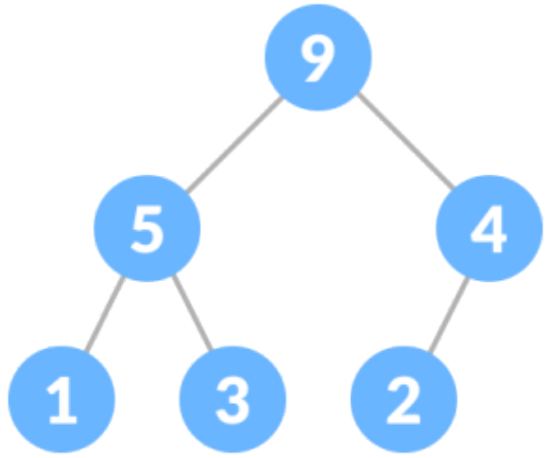
- Generally, heaps are of two orders/ types.
  - Min Heap
  - Max Heap

ARRAY BASED IMPLEMENTATIION

- Array-based representation of a binary heap, where for a node at index i:
- Left child is at 2i + 1
- Right child is at 2i + 2
- Parent is at (i - 1) / 2 (integer division)

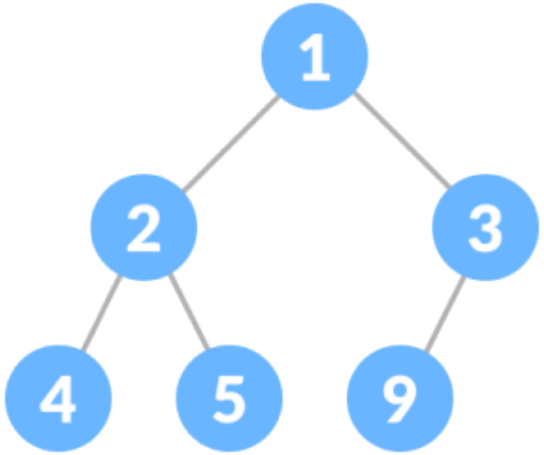
MAX HEAP

- In this heap, the value of the root node must be the greatest among all its child nodes and the same thing must be done for its left and right sub-tree also.



MIN HEAP

- In this heap, the value of the root node must be the smallest among all its child nodes and the same thing must be done for its left and right sub-tree also.



HEAP TREE CONSTRUCTION

- There can be two ways to create a heap tree:

    1. Insert each key one by one
        - Time complexity is O(n log n)
    2. Heapify Method
        - Time complexity is O(n)

HEAP CONSTRUCTION (ONE BY ONE)

- In this method, a heap tree is constructed according to the following points:
  - Insert the root node
  - Insert the next key as left child. After insertion compare it with its parent node
  - If value of parent node is greater than/ less than the value of left child then swap those
  - Insert the next key as right child. After insertion compare it with its parent node
  - If value of parent node is greater than/ less than the value of right child then swap those

HEAP CONSTRUCTION (ONE BY ONE)

- Insert key one by one in empty space takes O(1)
- To insert a key into already constructed heap takes O(log n) in worst case
- It takes log n comparisons and log n swapping
- If there are n elements total then it would be O(n log n)

HEAP CONSTRUCTION (ONE BY ONE)

- 14, 24, 12, 11, 25, 8, 35

ARRAY REPRESENTATION OF ABOVE HEAP

- [35, 24, 25, 11, 14, 8, 12]

HEAP CONSTRUCTION (HEAPIFY)

- 145, 40, 25, 65, 12, 48, 18, 1, 100, 27, 7, 3, 45, 9, 30

- We’ll use bottom-up heapify, starting from the last parent node:
- Last parent index = (n - 2) / 2 = (15 - 2)/2 = 6
- Apply heapify() from index 6 to 0.

1ST STEP

- Heapify at index 6: 18
- Children: index 13 (9) and 14 (30)
- Max child = 30 → 30 > 18 → swap

2ND STEP

- Heapify at index 5: 48
- Children: index 11 (3) and 12 (45)
- Max child = 45 → 45 < 48 → no change

3RD STEP

- Heapify at index 4: 12
- Children: index 9 (27) and 10 (7)
- Max child = 27 → 27 > 12 → swap

4TH STEP

- Heapify at index 3: 65
- Children: index 7 (1) and 8 (100)
- Max child = 100 → 100 > 65 → swap

FINAL HEAP

- [145, 100, 48, 65, 27, 45, 30, 1, 40, 12, 7, 3, 25, 9, 18]

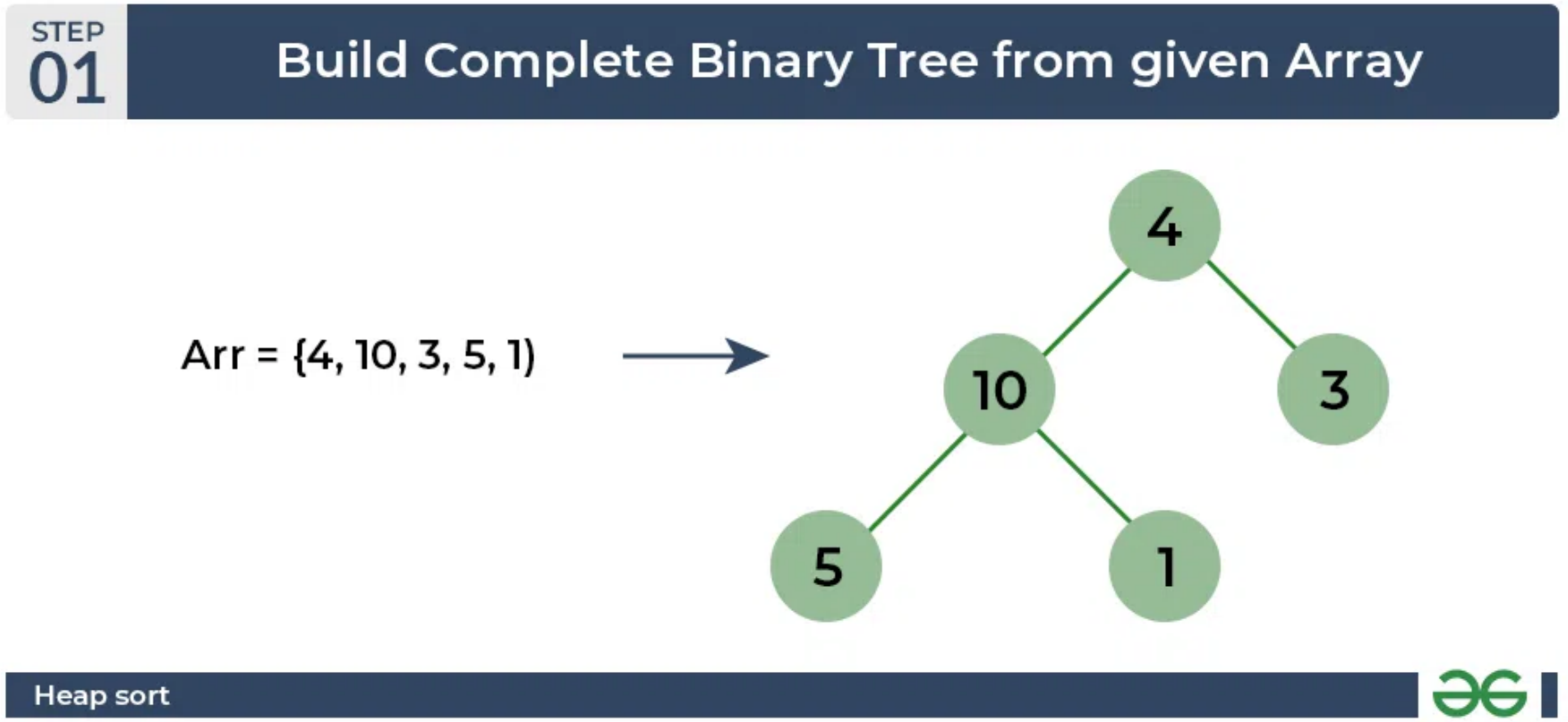
DELETION OF AN ELEMENT

- Deleting an element in heap tree is not simple
- You cannot delete an element directly from heap tree except that it is rightmost leaf element
- Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap
- Replace the root or element to be deleted by the last element
- Delete the last element from the Heap
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root
- Time complexity is O(log n) where n is no of elements in the heap

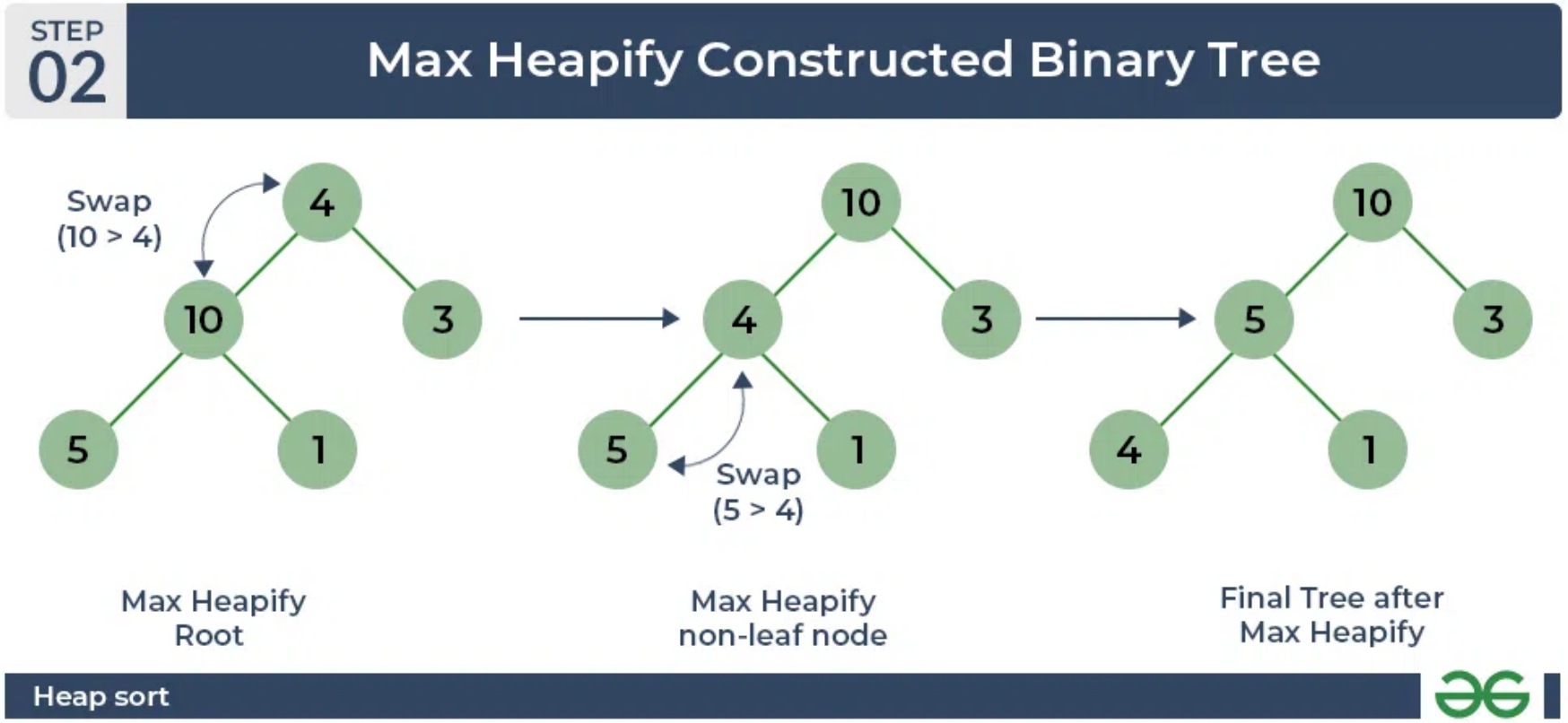
HEAP SORT

- Heap sort is a comparison-based sorting technique based on binary heap data structure.
- It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.
- In case of max heap:
  - One by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap.
- In case of min heap:
  - One by one delete the root node of the Min-heap and replace it with the first node in the heap and then heapify the root of the heap.
- Time Complexity: O(n log n) → O(n) + O(n log n)

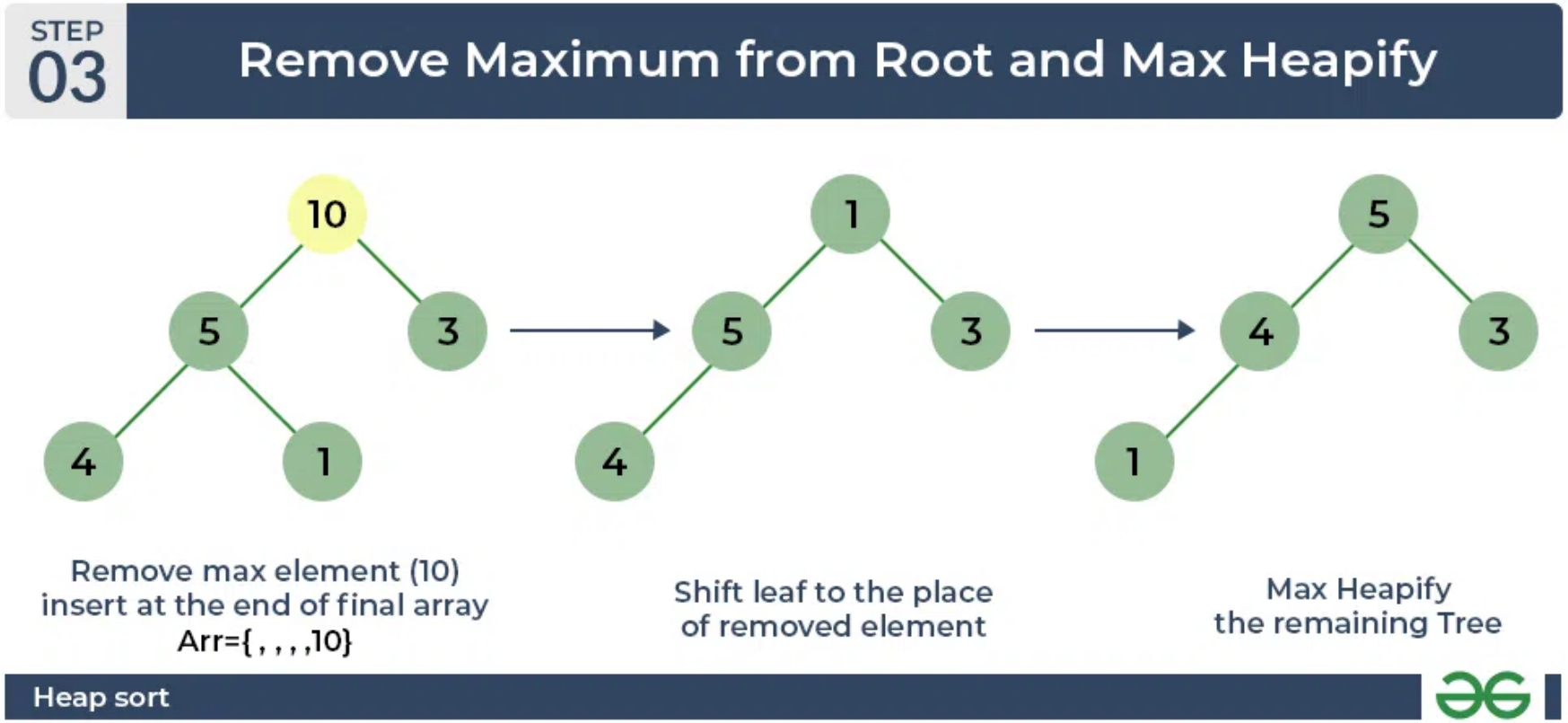
EXAMPLE: HEAP SORT



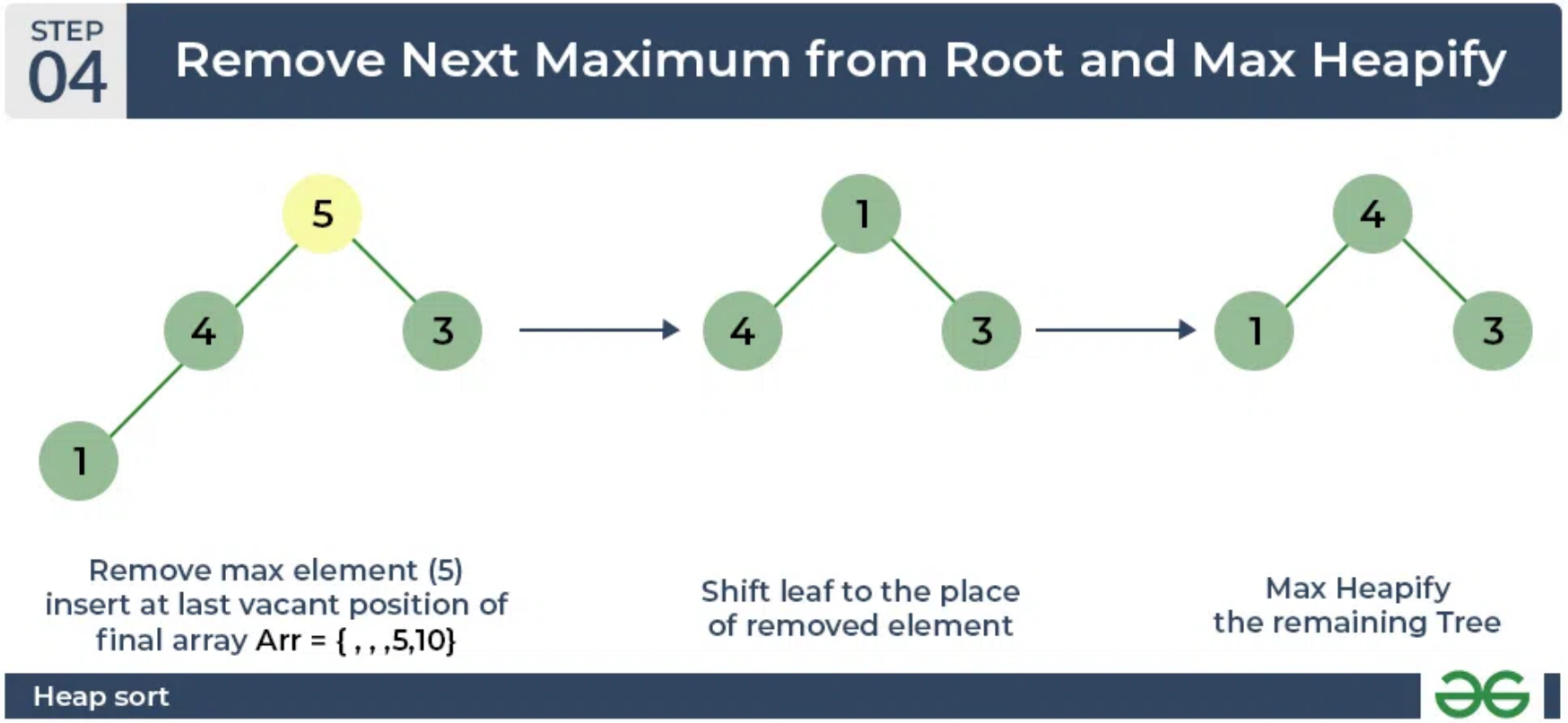
EXAMPLE: HEAP SORT



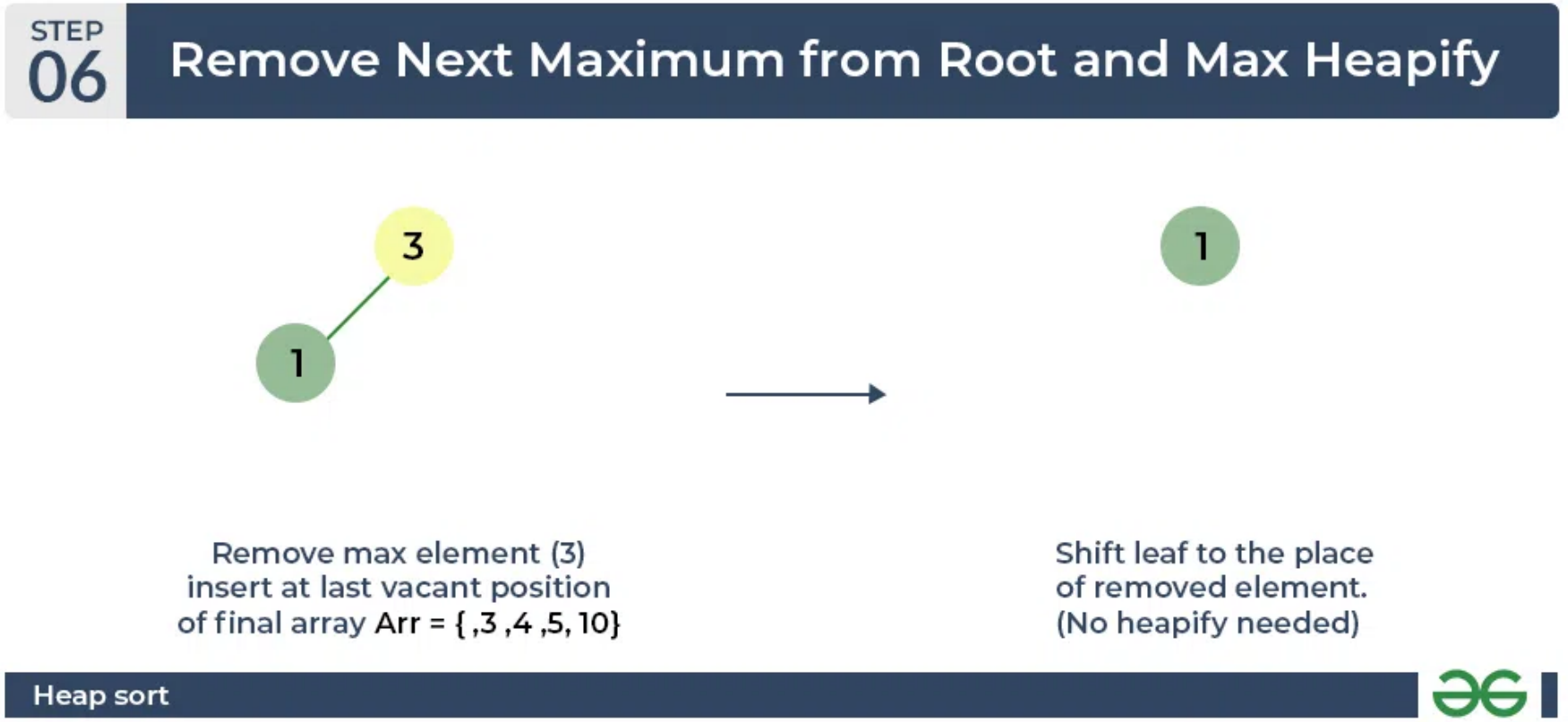
EXAMPLE: HEAP SORT



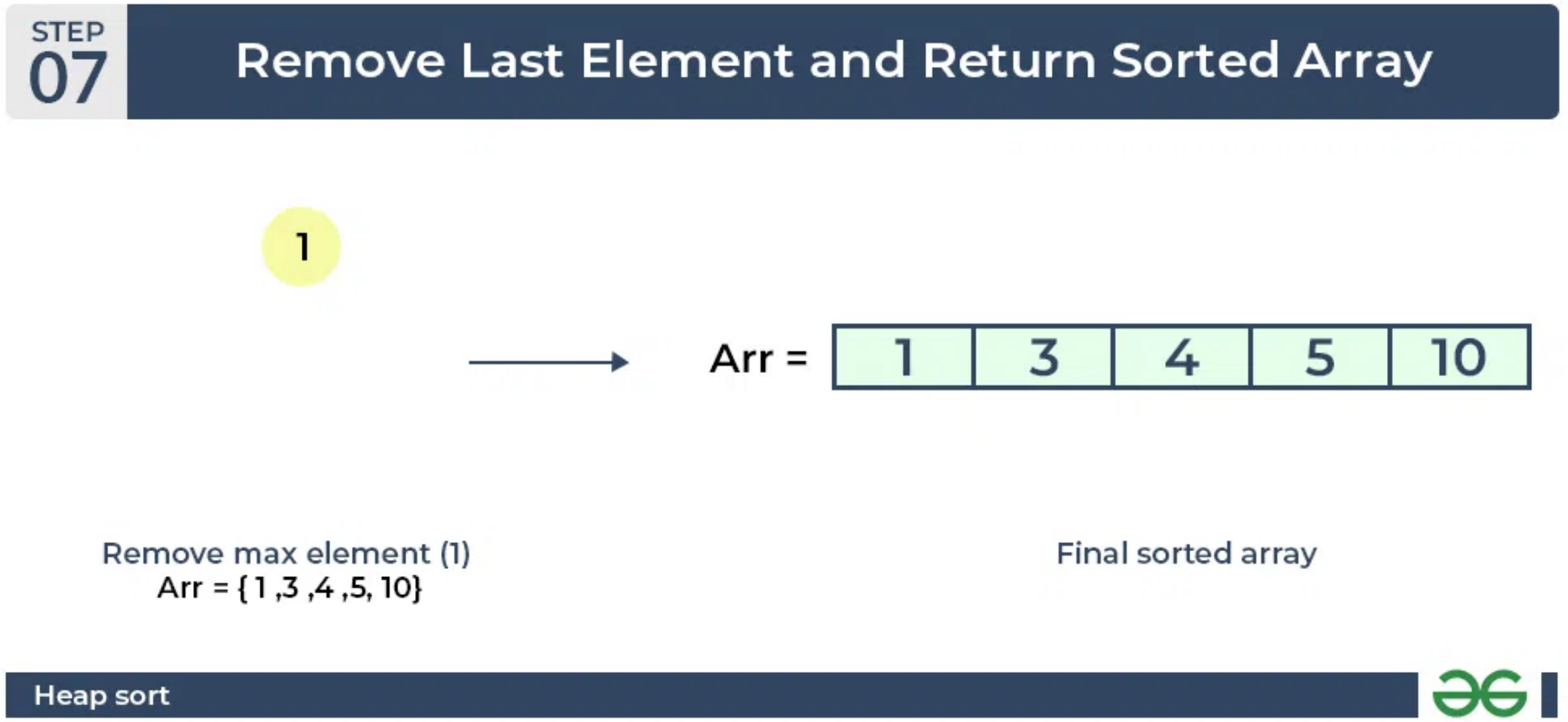
EXAMPLE: HEAP SORT



EXAMPLE: HEAP SORT



EXAMPLE: HEAP SORT

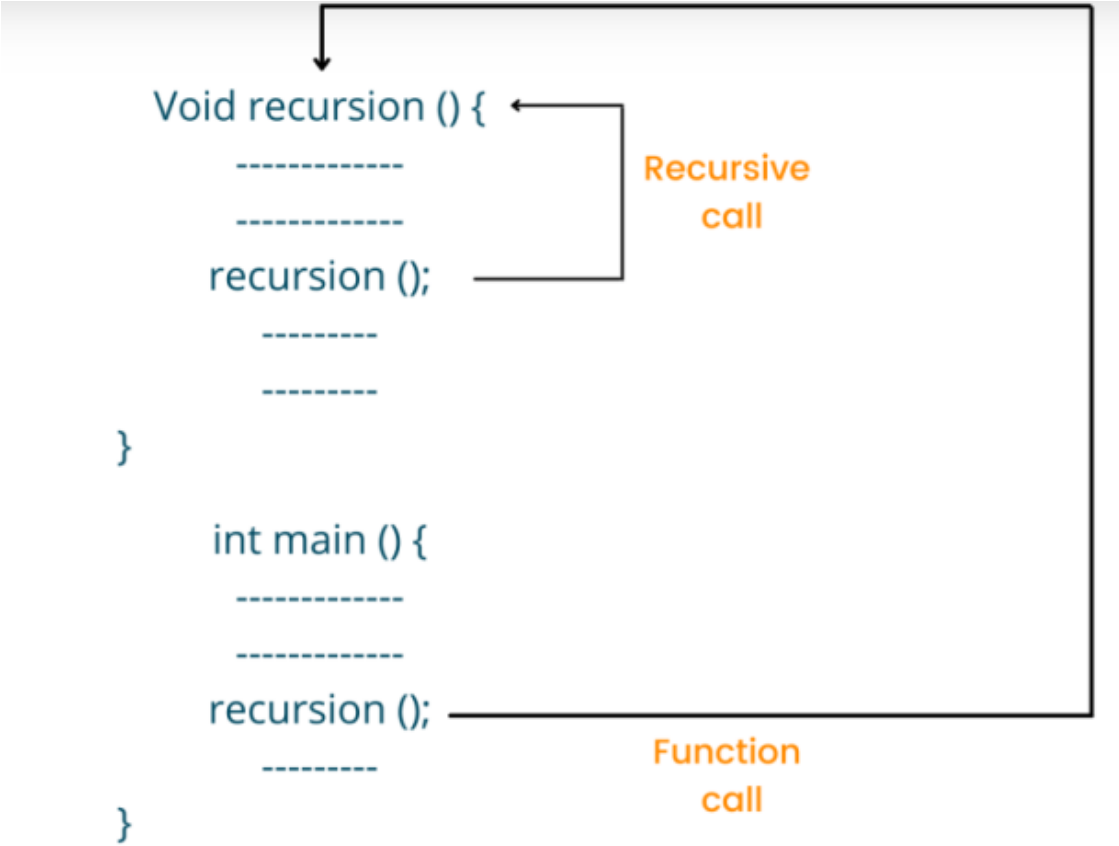


DATA STRUCTURES AND ALGORITHMS

RECURSION

- The process in which a function calls itself directly or indirectly is called recursion
- The corresponding function is called a recursive function.
- A recursive function solves a particular problem by calling a copy of itself
- Recursion has the ability of solving smaller subproblems of the original problems
- Many more recursive calls can be generated as and when required.
- It is essential to know that we should provide a certain case in order to terminate this recursion process
- So we can say that every time the function calls itself with a simpler version of the original problem

HOW RECURSION WORKS?



PROPERTIES OF RECURSION

- Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write.
- Performing the same operations multiple times with different inputs
- In every step, we try smaller inputs to make the problem smaller
- Base condition is needed to stop the recursion otherwise infinite loop will occur

ADVANTAGES OF RECURSION

- Less number code lines are used in the recursion program and hence the code looks shorter and cleaner.
- Recursion is easy to approach to solve the problems involving data structure and algorithms like graph and tree
- Recursion helps to reduce the time complexity
- It helps to reduce unnecessary calling of the function
- It helps to solve the stack evolutions and prefix, infix, postfix evaluation
- Recursion is the best method to define objects that have repeated structural forms

DISADVANTAGES OF RECURSION

- It consumes a lot of stack space
- It takes more time to process the program
- If an error is accrued in the program, it is difficult to debug the error in comparison to the iterative program.

RECURSION VS ITERATION

SR No.	Recursion	Iteration
1)	Terminates when the base case becomes true.	Terminates when the condition becomes false.
2)	Used with functions.	Used with loops.
3)	Every recursive call needs extra space in the stack memory.	Every iteration does not require any extra space.
4)	Smaller code size.	Larger code size.

HOW RECURSIVE FUNCTION ARE STORED IN MEMORY?

- Recursion uses more memory, because the recursive function adds to the stack with each recursive call
- It keeps the values there until the call is finished
- The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure

BASICS OF RECURSION

- There are two cases in the recursion/ recursive programs:
  - Base case
  - Recursive case
- In the recursive program, the solution to the base case is provided
- Base case terminates the recursive function
- Solution to the bigger problem is expressed in terms of smaller problems

FACTORIAL OF A NUMBER

Iterative Approach

```
for(i=1; i<=number; i++)
    fact=fact*i;
```

Recursive Approach

```
int fact(int n){
if(n > 1)
        return n*fact(n-1);
else
        return 1;
}
```

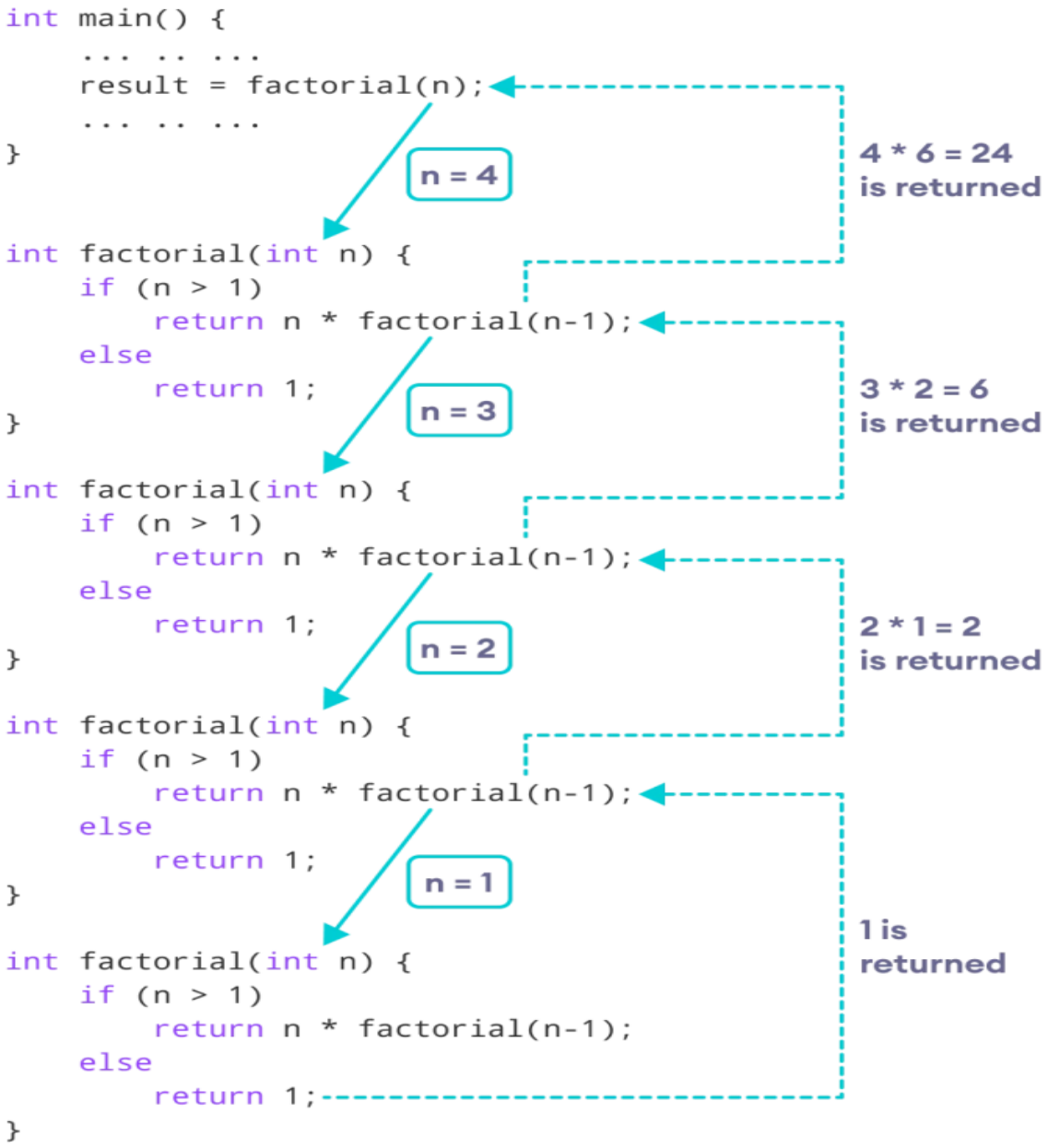
SUM OF NUMBERS FROM 1-5

Iterative Approach

```
int main()

{
  int result = 0;

 for(i=1; i<= 5; i++)

       result = result + 1;

cout<<result;

}
```

Recursive Approach

```
int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}

int main() {
  int result = sum(5);
  cout << result;
  return 0;
}
```


int main() {
... .
result = factorial(n);
... .
}
n = 4
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 3
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 2
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 1
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
4 * 6 = 24
is returned
3 * 2 = 6
is returned
2 * 1 = 2
is returned
1 is
returned

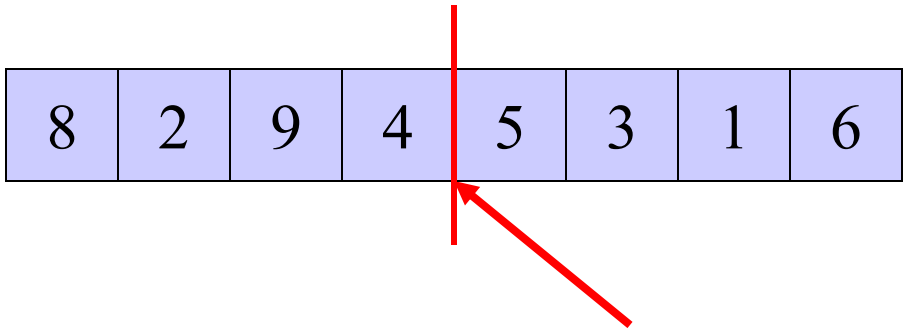
Sorting (Part II: Divide and Conquer)

Data Structures and Algorithm

“Divide and Conquer”

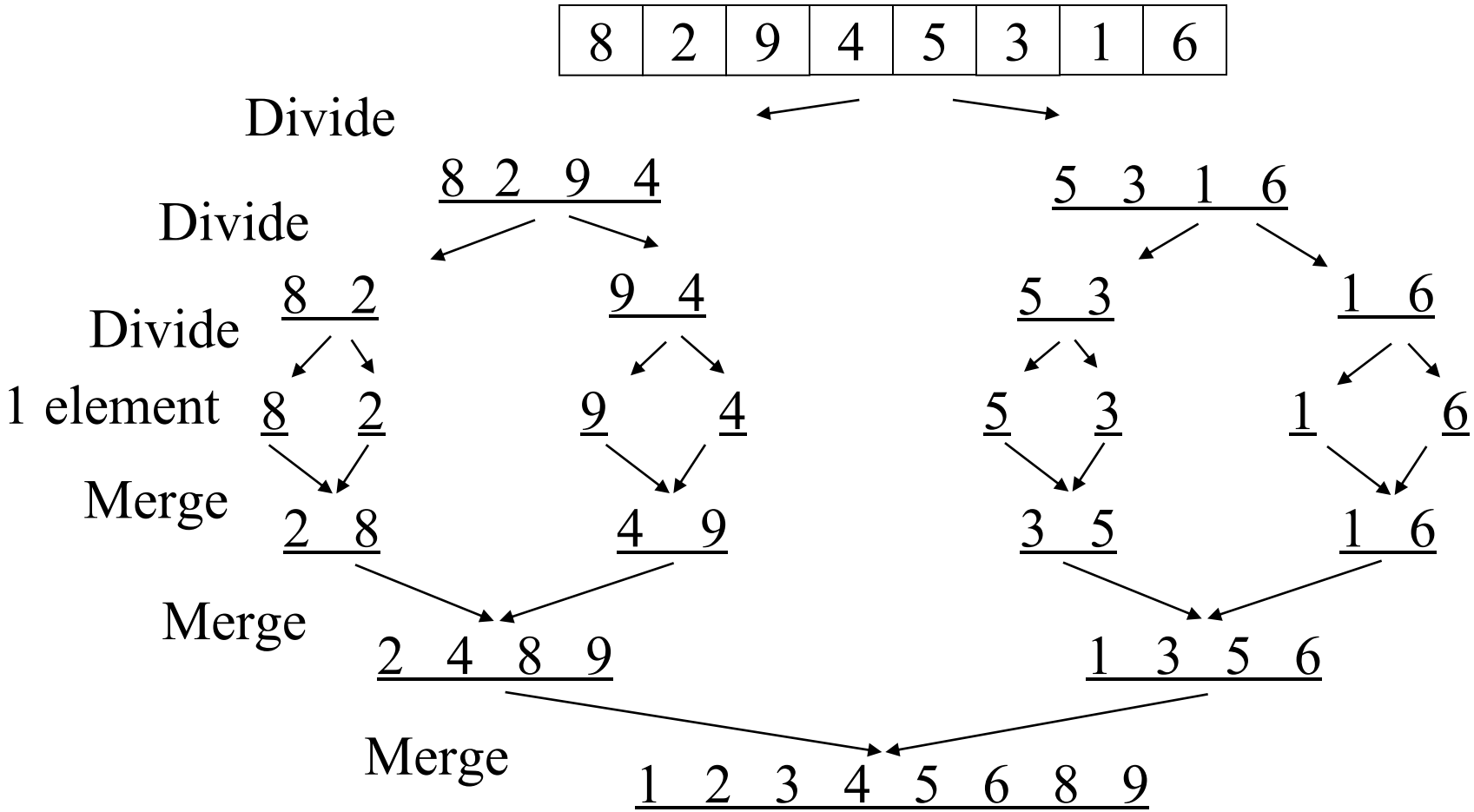
- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort
- **Idea 2 :** Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → Quicksort

Mergesort



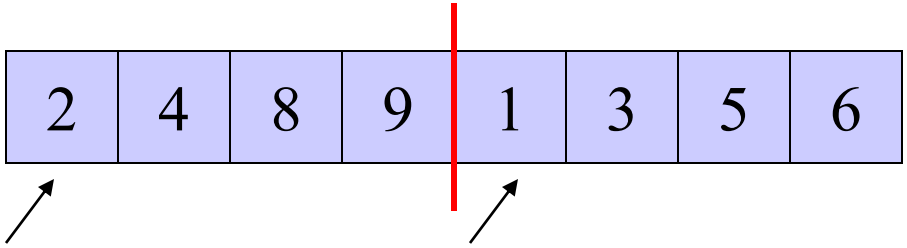
- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

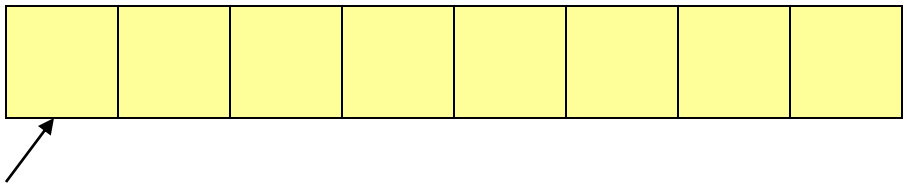
Mergesort Example



Auxiliary Array

- The merging requires an auxiliary array.

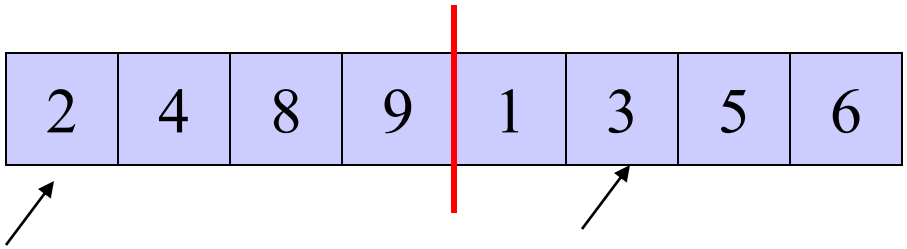


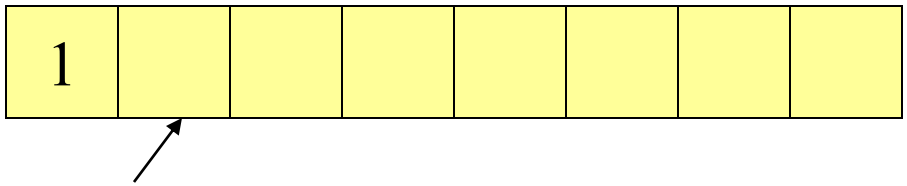


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

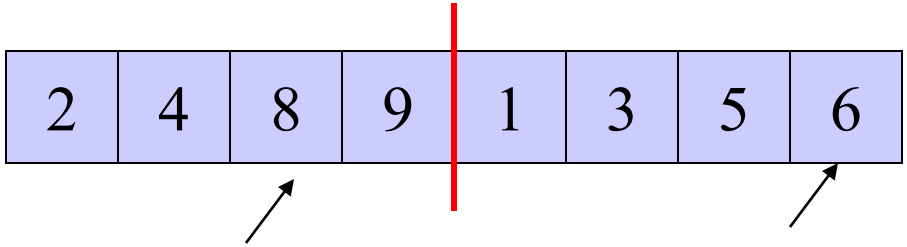


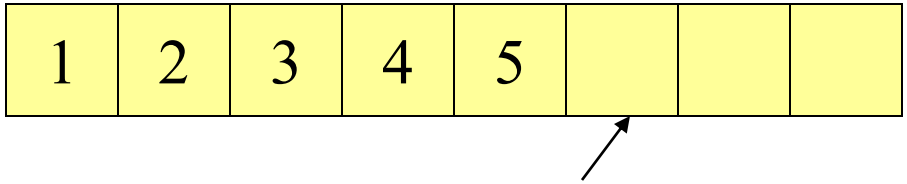


Auxiliary array

Auxiliary Array

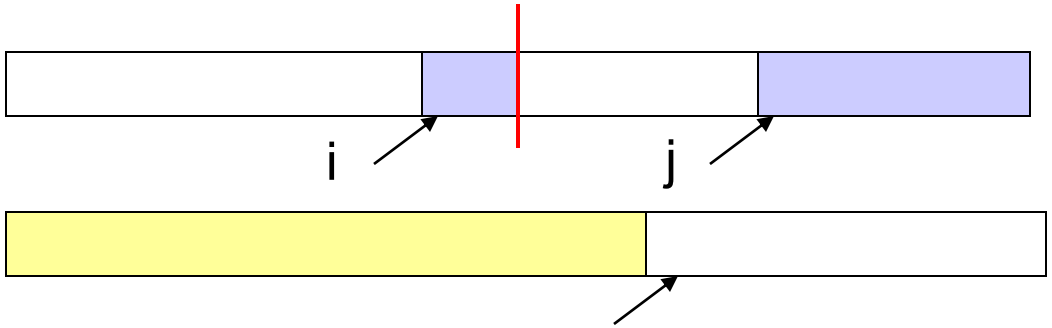
- The merging requires an auxiliary array.



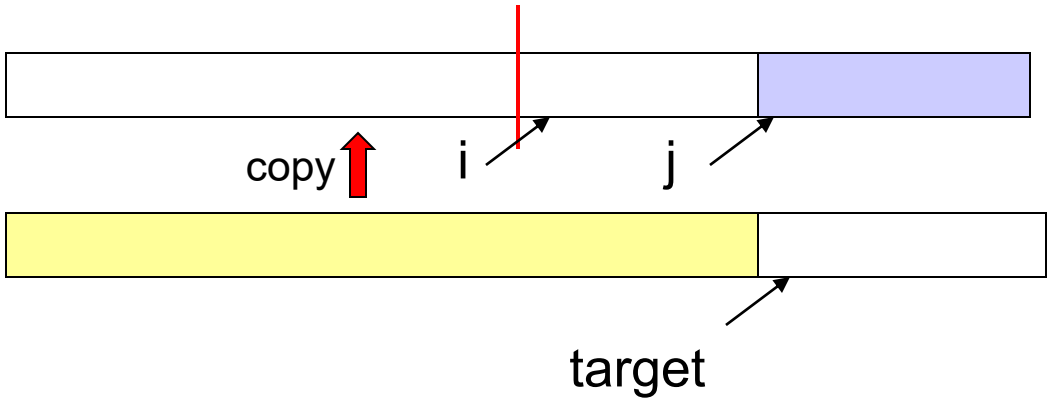


Auxiliary array

Merging

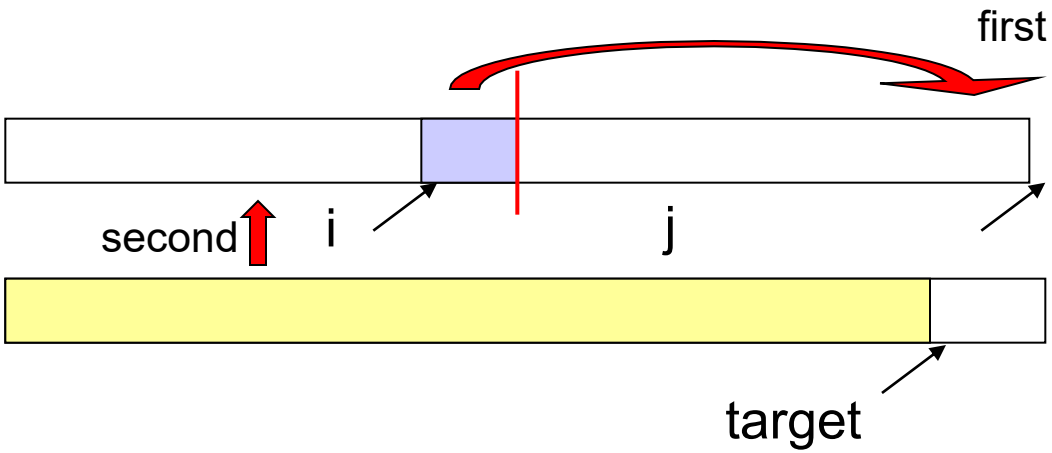


normal



Left completed first

Merging



Right completed first

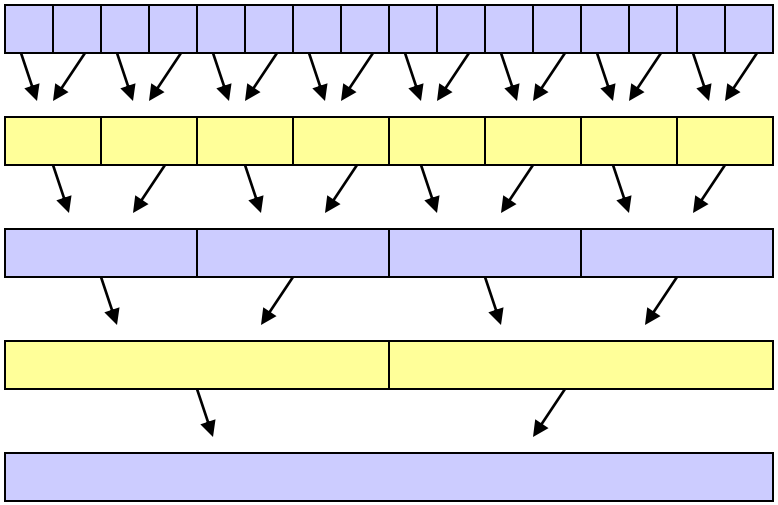
Merging

```
Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i ≤ mid and j ≤ right do
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
      else T[target] := A[j]; j := j + 1;
    target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k : = mid; l := right;
    while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
```

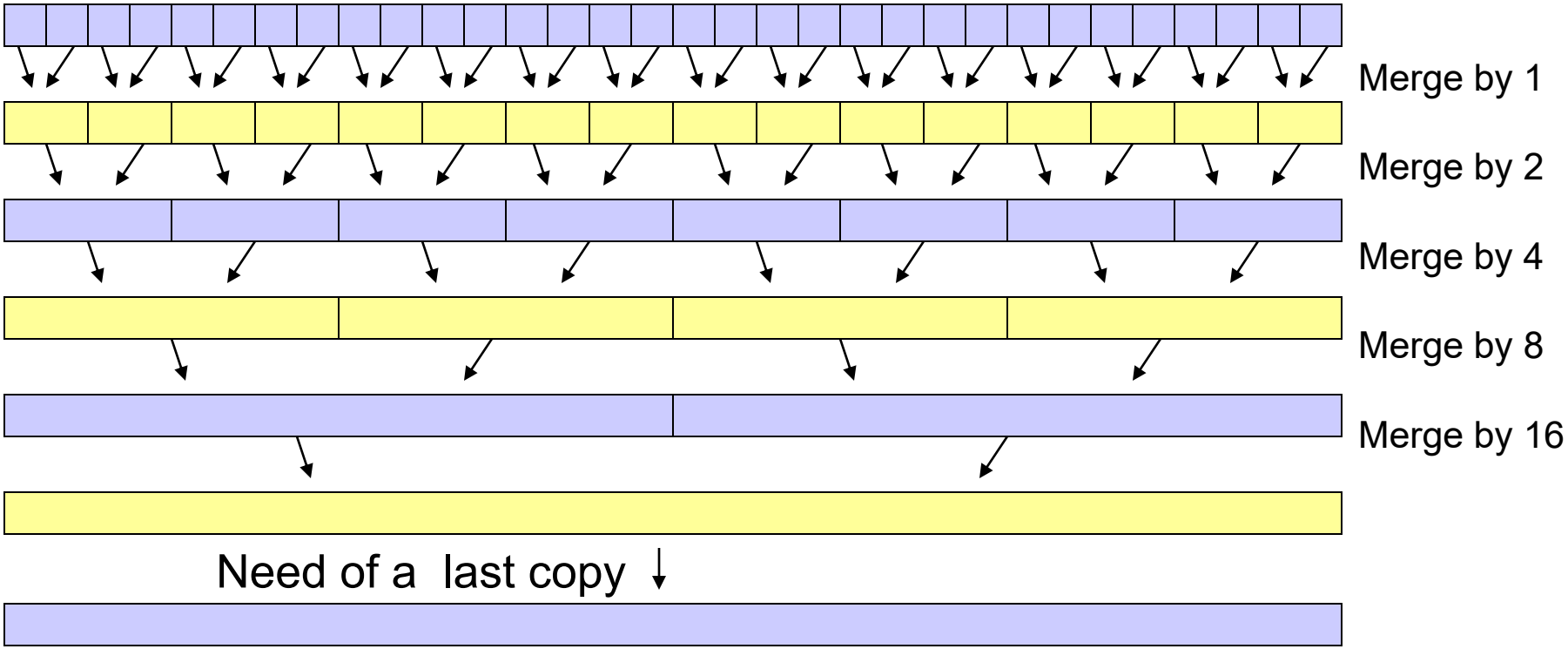
Recursive Mergesort

```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}
```

Iterative Mergesort



Iterative Mergesort



Iterative Mergesort

```
IterativeMergesort(A[1..n]: integer array, n : integer) : {
//precondition: n is a power of 2//
  i, m, parity : integer;
  T[1..n]: integer array;
  m := 2; parity := 0;
  while m ≤ n do
    for i = 1 to n - m + 1 by m do
       if parity = 0 then Merge(A,T,i,i+m-1);
         else Merge(T,A,i,i+m-1);
    parity := 1 - parity;
    m := 2*m;
  if parity = 1 then
    for i = 1 to n do A[i] := T[i];
}
```

How do you handle non-powers of 2?
How can the final copy be avoided?

Mergesort Analysis

- Let T(N) be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes T(N/2) and merging takes O(N)

Mergesort Recurrence Relation

- The recurrence relation for T(N) is:
  - $T(1) \leq a$
    - base case: 1 element array → constant time
  - $T(N) \leq 2T(N/2) + bN$
    - Sorting N elements takes
      - the time to sort the left half
      - plus the time to sort the right half
      - plus an O(N) time to merge the two halves
- $T(N) = O(n \log n)$

Properties of Mergesort

- Not in-place
  - › Requires an auxiliary array (O(n) extra space)
- Stable
  - › Make sure that left is sent to target on equal values.
- Iterative Mergesort reduces copying.

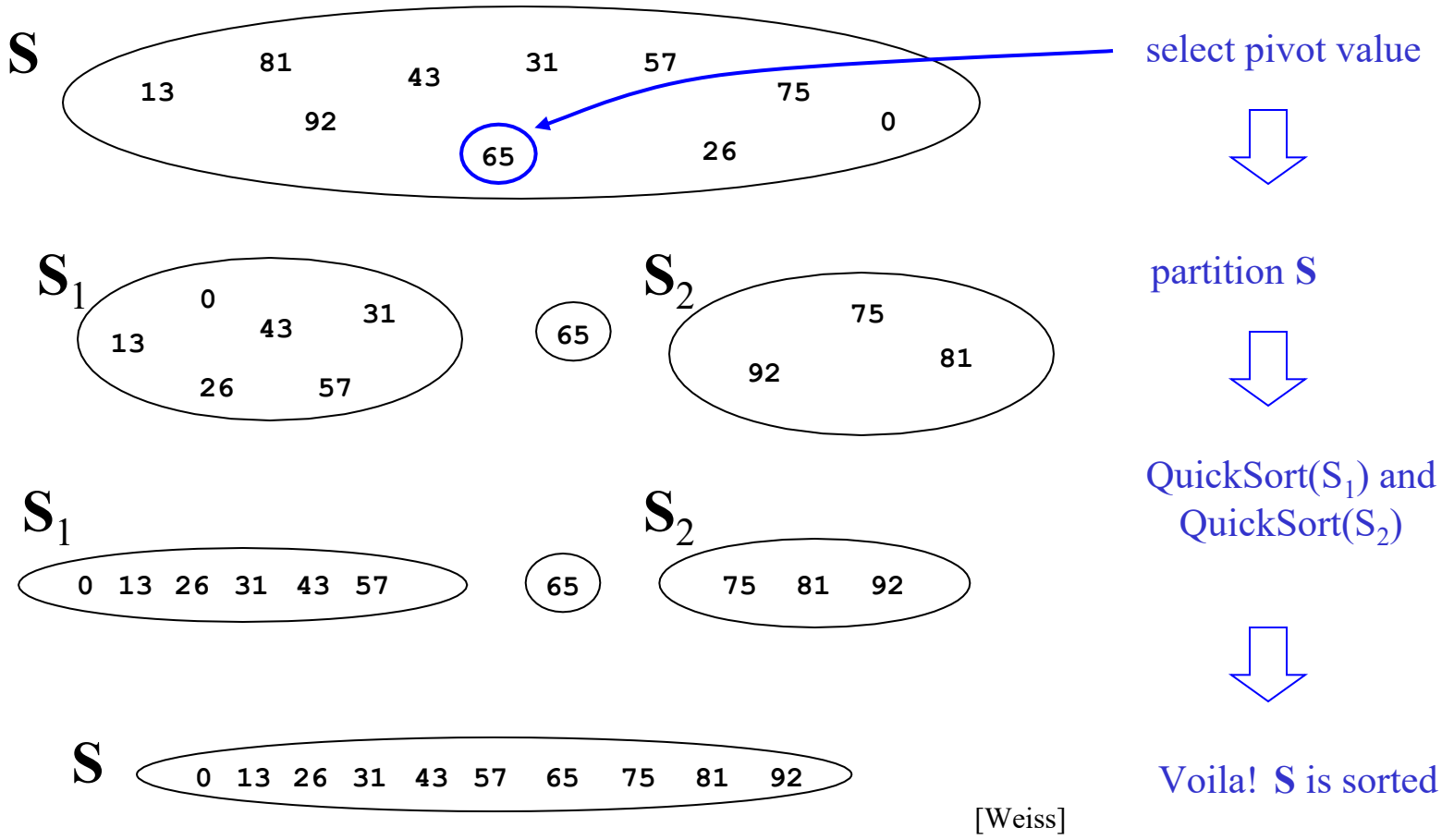
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

"Four easy steps"

- To sort an array **S**
  1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  2. Pick an element *v* in **S**. This is the *pivot* value.
  3. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \le v$}, and $\mathbf{S}_2$ = {all values $x \ge v$}.
  4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

The steps of QuickSort



[Weiss]

Details

- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are $\leq$ pivot
  - elements in right sub-array are $\geq$ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning:Choosing the pivot

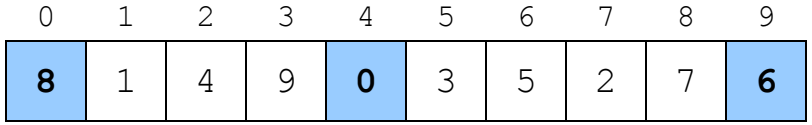
- One implementation (there are others)
  - › median3 finds pivot and sorts left, center, right
    - Median3 takes the median of leftmost, middle, and rightmost elements
    - An alternative is to choose the pivot randomly (need a random number generator; “expensive”)
    - Another alternative is to choose the first element (but can be very bad. Why?)
  - › Swap pivot with next to last element

Partitioning in-place

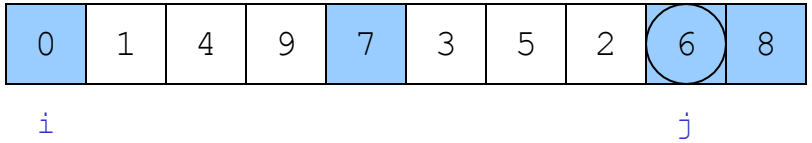
- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit elmt A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-2]) with A[i]

Example

Choose the pivot as the median of three

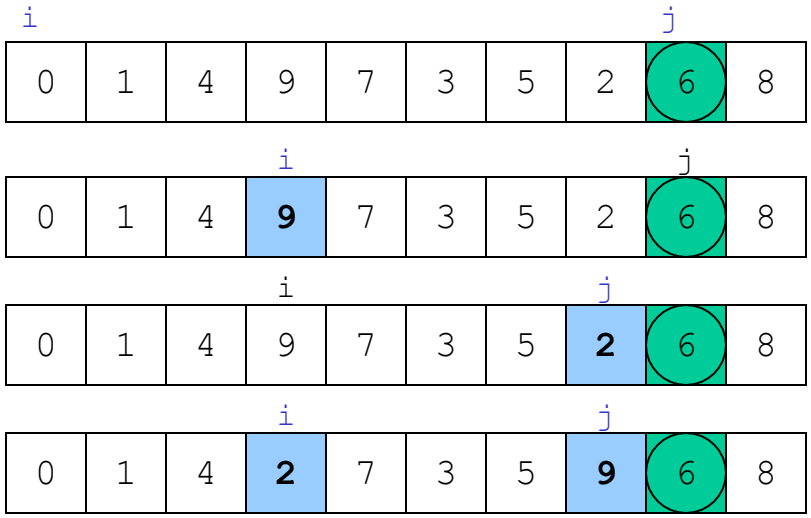


Median of 0, 6, 8 is 6. Pivot is 6



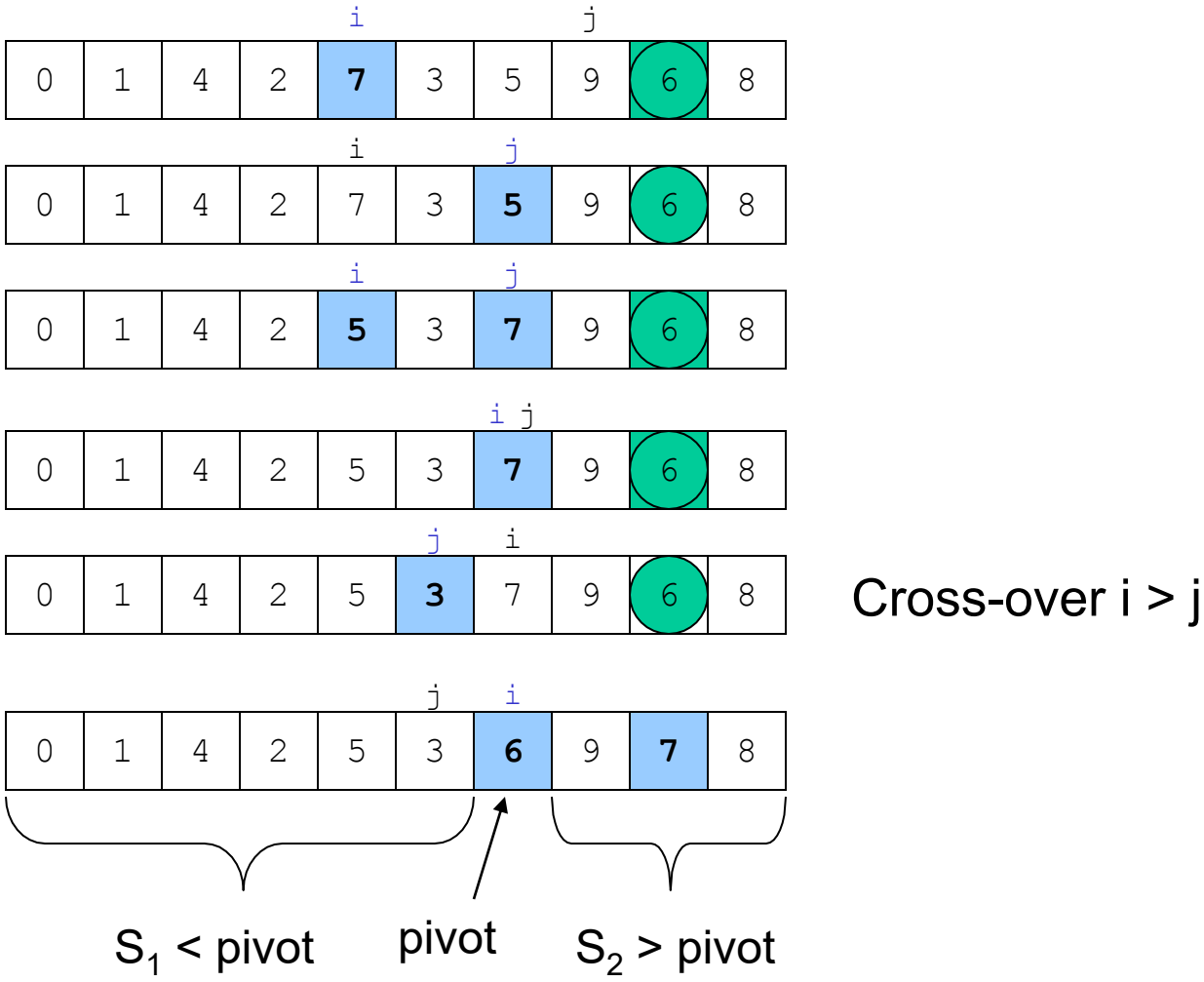
Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example



Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap

Example



Cross-over i > j

Recursive Quicksort

```
Quicksort(A[]: integer array, left,right : integer): {
pivotindex : integer;
if left + CUTOFF ≤ right then
  pivot := median3(A,left,right);
  pivotindex := Partition(A,left,right-1,pivot);
  Quicksort(A, left, pivotindex – 1);
  Quicksort(A, pivotindex + 1, right);
else
  Insertionsort(A,left,right);
}
```

Don’t use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - › T(0) = T(1) = O(1)
    - constant time if 0 or 1 element
  - › For N > 1, 2 recursive calls plus linear time for partitioning
  - › T(N) = 2T(N/2) + O(N)
    - Same recurrence relation as Mergesort
  - › T(N) = O(N log N)

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - $T(N) \leq a$ for $N \leq C$
  - $T(N) \leq T(N-1) + bN$
  - $\leq T(N-2) + b(N-1) + bN$
  - $\leq T(C) + b(C+1) + \ldots + bN$
  - $\leq a + b(C + (C+1) + (C+2) + \ldots + N)$
  - $T(N) = O(N^2)$
- Fortunately, *average case performance* is O(N log N) (see text for proof)

Properties of Quicksort

- Not stable because of long distance swapping.
- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive call (O(logn) space).
- O(n log n) average case performance, but O (n^2) worst case performance.

Folklore

- “Quicksort is the best in-memory sorting algorithm.”
- Truth
  - Quicksort uses very few comparisons on average.
  - Quicksort does have good performance in the memory hierarchy.

```
class AVLNode {
    String key;
    int height;
    AVLNode left, right;

    AVLNode(String d) {
        key = d;
        height = 1;
    }
}

public class AVLTree {
    AVLNode root;

    //Get height of node
    int height(AVLNode N) {
        return (N == null) ? 0 : N.height;
    }

    //Get balance factor
    int getBalance(AVLNode N) {
        return (N == null) ? 0 : height(N.left) - height(N.right);
    }

    //Right rotation
    AVLNode rightRotate(AVLNode y) {
        AVLNode x = y.left;
        AVLNode T2 = x.right;

        x.right = y;
        y.left = T2;

        y.height = Math.max(height(y.left), height(y.right)) + 1;
        x.height = Math.max(height(x.left), height(x.right)) + 1;

        return x;
    }

    //Left rotation
    AVLNode leftRotate(AVLNode x) {
        AVLNode y = x.right;
        AVLNode T2 = y.left;

        y.left = x;
        x.right = T2;

        x.height = Math.max(height(x.left), height(x.right)) + 1;
        y.height = Math.max(height(y.left), height(y.right)) + 1;

        return y;
    }

    //AVL Insert with balancing
    AVLNode insert(AVLNode node, String key) {
```

```
    if (node == null)
        return new AVLNode(key);

    if (key.compareTo(node.key) < 0)
        node.left = insert(node.left, key);
    else if (key.compareTo(node.key) > 0)
        node.right = insert(node.right, key);
    else
        return node; // Duplicate not allowed

    // Update height
    node.height = 1 + Math.max(height(node.left), height(node.right));

    // Get balance factor
    int balance = getBalance(node);

    //Perform rotations

    // Left Left Case
    if (balance > 1 && key.compareTo(node.left.key) < 0)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key.compareTo(node.right.key) > 0)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key.compareTo(node.left.key) > 0) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key.compareTo(node.right.key) < 0) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}

//In-order traversal (Alphabetical order)
void inOrderTraversal(AVLNode node) {
    if (node != null) {
        inOrderTraversal(node.left);
        System.out.print(node.key + " â†’ ");
        inOrderTraversal(node.right);
    }
}

// â¬‡ï¸ Wrapper method
void addStudent(String name) {
    root = insert(root, name);
}
```

```
    void displayStudents() {
        System.out.println("Students in alphabetical order:");
        inOrderTraversal(root);
        System.out.println("null");
    }

    //To be completed by students
    void deleteStudent(String name) {
        // TODO: Implement delete method with rebalancing
    }

    boolean findStudent(String name) {
        // TODO: Implement search method
        return false;
    }

    //Main method to test
    public static void main(String[] args) {
        AVLTree archive = new AVLTree();

        archive.addStudent("Mango");
        archive.addStudent("Apple");
        archive.addStudent("Banana");
        archive.addStudent("Peach");
        archive.addStudent("Orange");

        archive.displayStudents();

        // â□Œ Uncomment once students complete these
        // System.out.println("Found Mango? " +
archive.findStudent("Mango"));
        // archive.deleteStudent("Banana");
        // archive.displayStudents();
    }
}
```

```
public Node insert(Node root, int key) {
    if (root == null) {
        return new Node(key);
    }
    if (key < root.data) {
        root.left = insert(root.left, key);
    } else if (key > root.data) {
        root.right = insert(root.right, key);
    }
    return root;
}


public boolean search(Node root, int key) {
    if (root == null) return false;
    if (root.data == key) return true;
    if (key < root.data)
        return search(root.left, key);
    else
        return search(root.right, key);
}



public Node delete(Node root, int key) {
    if (root == null) return null;

    if (key < root.data) {
        root.left = delete(root.left, key);
    } else if (key > root.data) {
        root.right = delete(root.right, key);
    } else {
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        root.data = minValue(root.right);
        root.right = delete(root.right, root.data);
    }
    return root;
}

private int minValue(Node root) {
    int minv = root.data;
    while (root.left != null) {
        minv = root.left.data;
        root = root.left;
    }
    return minv;
}


public void inorder(Node root) {
```

```
    if (root != null) {
        inorder(root.left);
        System.out.print(root.data + " ");
        inorder(root.right);
    }
}


public void preorder(Node root) {
    if (root != null) {
        System.out.print(root.data + " ");
        preorder(root.left);
        preorder(root.right);
    }
}


public void postorder(Node root) {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.data + " ");
    }
}
```

```
public void dfs(int v, boolean[] visited, List<List<Integer>> adj) {
    visited[v] = true;
    System.out.print(v + " ");
    for (int neighbor : adj.get(v)) {
        if (!visited[neighbor]) {
            dfs(neighbor, visited, adj);
        }
    }
}

public void bfs(int start, List<List<Integer>> adj) {
    boolean[] visited = new boolean[adj.size()];
    Queue<Integer> queue = new LinkedList<>();

    visited[start] = true;
    queue.add(start);

    while (!queue.isEmpty()) {
        int v = queue.poll();
        System.out.print(v + " ");
        for (int neighbor : adj.get(v)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}



List<List<Integer>> adj = new ArrayList<>();
int vertices = 5;
for (int i = 0; i < vertices; i++) {
    adj.add(new ArrayList<>());
}

// Add edges
adj.get(0).add(1);
adj.get(0).add(2);
adj.get(1).add(3);
adj.get(2).add(4);
```

```
public void HashSet() {
    HashSet<String> set = new HashSet<>();

    set.add("apple");
    set.add("banana");
    if (set.contains("apple")) {
        System.out.println("Apple is in the set");
    }
    for (String item : set) {
        System.out.println(item);
    }
}
```

```
public void HashMap() {
    HashMap<String, Integer> map = new HashMap<>();

    map.put("apple", 3);
    map.put("banana", 2);

    int count = map.get("apple");

    if (map.containsKey("banana")) {
        System.out.println("Banana exists");
    }
    for (Map.Entry<String, Integer> entry : map.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
    }
}
```

```
public void heapSort(int[] arr) {
    int n = arr.length;

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

private void heapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        heapify(arr, n, largest);
    }
}
```

Matrix Representation:

```
int[][] adjMatrix = new int[vertices][vertices];

// Add edge from u to v
adjMatrix[u][v] = 1;

// For undirected graph, also add reverse edge
adjMatrix[v][u] = 1;
```

List Representation:

```
List<List<Integer>> adjList = new ArrayList<>();

for (int i = 0; i < vertices; i++) {
    adjList.add(new ArrayList<>());
}

// Add edge from u to v
adjList.get(u).add(v);

// For undirected graph, also add reverse edge
adjList.get(v).add(u);
```

```
public class MaxHeap {
    private int[] heap;
    private int size;

    public MaxHeap(int capacity) {
        heap = new int[capacity];
        size = 0;
    }

    public void insert(int value) {
        heap[size] = value;
        int current = size;
        while (current > 0 && heap[current] > heap[parent(current)]) {
            swap(current, parent(current));
            current = parent(current);
        }
        size++;
    }

    public int removeMax() {
        if (size == 0) return -1;
        int max = heap[0];
        heap[0] = heap[--size];
        maxHeapify(0);
        return max;
    }

    private void maxHeapify(int i) {
        int left = leftChild(i);
        int right = rightChild(i);
        int largest = i;

        if (left < size && heap[left] > heap[largest]) largest = left;
        if (right < size && heap[right] > heap[largest]) largest = right;

        if (largest != i) {
            swap(i, largest);
            maxHeapify(largest);
        }
    }

    private int parent(int i) { return (i - 1) / 2; }
    private int leftChild(int i) { return 2 * i + 1; }
    private int rightChild(int i) { return 2 * i + 2; }
    private void swap(int i, int j) {
        int temp = heap[i]; heap[i] = heap[j]; heap[j] = temp;
    }
}
```

```
public void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

private void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
public class MinHeap {
    private int[] heap;
    private int size;

    public MinHeap(int capacity) {
        heap = new int[capacity];
        size = 0;
    }

    public void insert(int value) {
        heap[size] = value;
        int current = size;
        while (current > 0 && heap[current] < heap[parent(current)]) {
            swap(current, parent(current));
            current = parent(current);
        }
        size++;
    }

    public int removeMin() {
        if (size == 0) return -1;
        int min = heap[0];
        heap[0] = heap[--size];
        minHeapify(0);
        return min;
    }

    private void minHeapify(int i) {
        int left = leftChild(i);
        int right = rightChild(i);
        int smallest = i;

        if (left < size && heap[left] < heap[smallest]) smallest = left;
        if (right < size && heap[right] < heap[smallest]) smallest =
right;

        if (smallest != i) {
            swap(i, smallest);
            minHeapify(smallest);
        }
    }

    private int parent(int i) { return (i - 1) / 2; }
    private int leftChild(int i) { return 2 * i + 1; }
    private int rightChild(int i) { return 2 * i + 2; }
    private void swap(int i, int j) {
        int temp = heap[i]; heap[i] = heap[j]; heap[j] = temp;
    }
}
```

```
public void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

private int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
```

TreeMap is a part of the Java Collection Framework. It implements the Map and NavigableMap interface and extends the AbstractMap class. It stores key-value pairs in a sorted order based on the natural ordering of keys or a custom Comparator. It uses a Red-Black Tree for efficient operations (add, remove, retrieve) with a time complexity of O(log n).

The keys in a TreeMap are always sorted.
Most operations, such as get, put, and remove have a time complexity of O(log n).
TreeMap does not allow null as a key, it allows null as a value. Attempting to insert a null key will result in NullPointerException.
TreeMap is not Synchronized. For thread-safe operations, we need to use Collections.synchronized map.
Entry pairs returned by the methods in this class and their views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.

```
// Java Program to create a TreeMap
import java.util.Map;
import java.util.TreeMap;

public class TreeMapCreation {
    public static void main(String args[])
    {
        // Create a TreeMap of Strings
        // (keys) and Integers (values)
        TreeMap<String, Integer> tm = new TreeMap<>();

        System.out.println("TreeMap elements: " + tm);
    }
}
```

```
// Performing basic operations on TreeMap
import java.util.Map;
import java.util.TreeMap;

public class Geeks {
    public static void main(String[] args)
    {
        Map<String, Integer> tm = new TreeMap<>();

        // Adding elements to the tree map
        tm.put("A", 1);
        tm.put("C", 3);
        tm.put("B", 2);

        // Getting values from the tree map
        int ans = tm.get("A");
        System.out.println("Value of A: " + ans);

        // Removing elements from the tree map
        tm.remove("B");

        // Iterating over the elements of the tree map
```

```
        for (String key : tm.keySet()) {
            System.out.println("Key: " + key + ", Value: "
                               + tm.get(key));
        }
    }
}
```

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree(red - black tree) for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.

TreeSet does not allow duplicate elements. Any attempt to add a duplicate element will be ignored.
It doesn't allow null values and throws NullPointerException null element is inserted in it.
TreeSet implements the NavigableSet interface and provides additional methods to navigate the set (e.g., higher(), lower(), ceiling(), and floor()).

```
// Java Program Implementing TreeSet
import java.util.TreeSet;

public class TreeSetCreation
{
    public static void main(String args[])
    {
        // Create a TreeSet of Strings
        TreeSet<String> t = new TreeSet<>();

        // Displaying the TreeSet (which is empty at this point)
        System.out.println("TreeSet elements: " + t);
    }
}
```

```
public void TreeSet() {
    TreeSet<Integer> treeSet = new TreeSet<>();

    treeSet.add(5);
    treeSet.add(1);
    treeSet.add(10);

    for (int num : treeSet) {
        System.out.print(num + " ");
    }

    System.out.println("Min: " + treeSet.first());
    System.out.println("Max: " + treeSet.last());
}
```