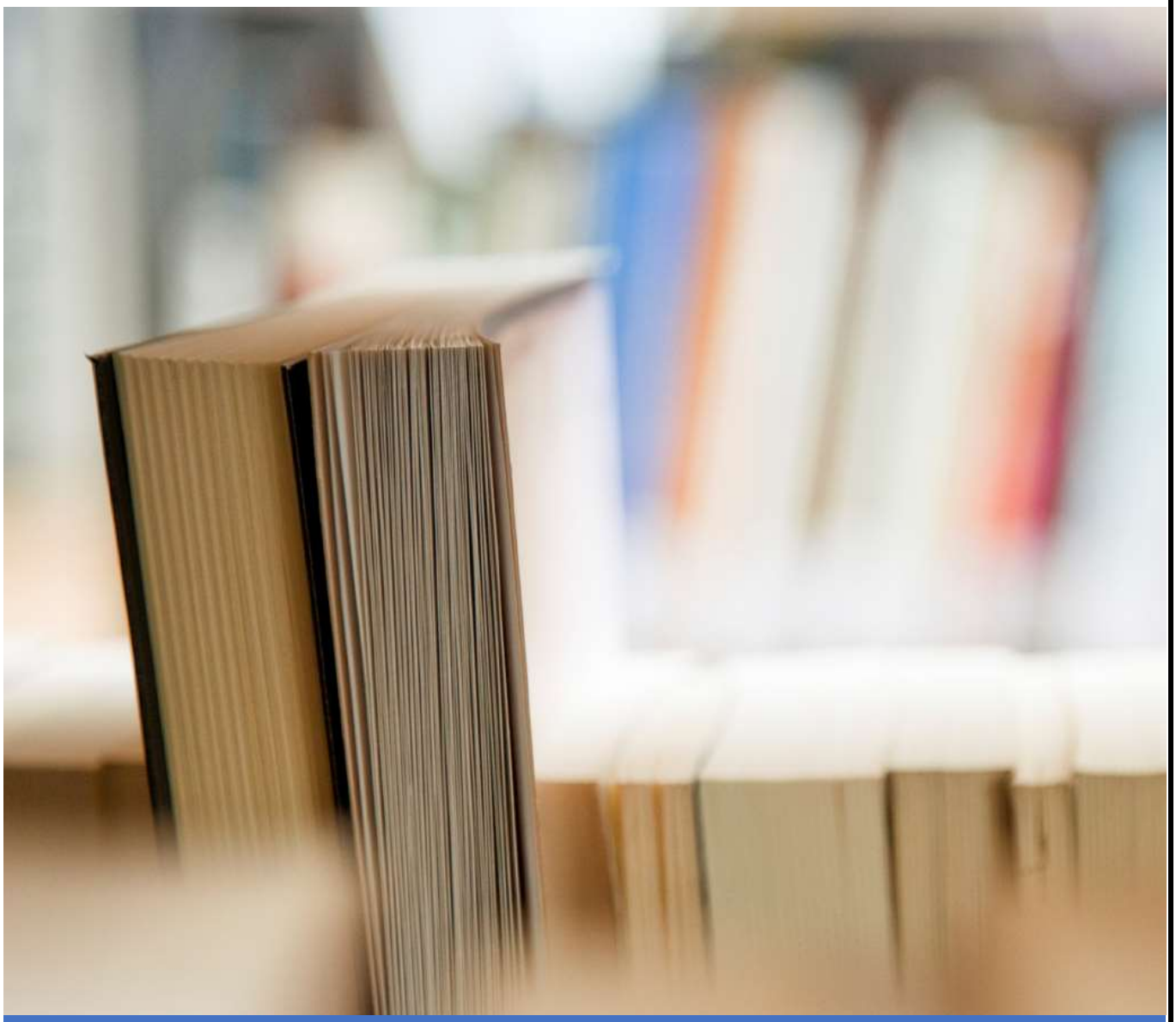


Data Structure & Algorithms

Notes

Ashirbad swain

8/31/20

Data Structure

1. Data Structure Introduction
2. Algorithm Introduction
3. Asymptotic Notation
4. Stacks
    - Stack Operations
    - Applications of Stack
    - Advantages & Disadvantages of stack
5. Queue
    - Queue Operations
    - Applications of Queue
    - Advantages & Disadvantages of queue
6. Arrays
    - Array Operations
    - Applications of arrays
    - Advantages & Disadvantages of array
7. Linked Lists
    - Linked list Operations
    - Applications of Linked List
    - Advantages & disadvantages of linked list
8. Hash Tables
    - Hash Function
    - Applications of hash table
    - Advantages & Disadvantages of hash table
9. Trees
    - Binary Search Tree
    - Applications of trees
    - Advantages & Disadvantages of trees
10. Heaps
    - Applications of heaps
11. Graphs
    - Directed Graph
    - Undirected Graph
    - Applications of graph
12. Sort Algorithms
13. Search Algorithms
14. Recursion
15. Dynamic Programming
16. Master Theorem
17. Divide & Conquer Method

Data Structure Introduction

- Data Structure is a format for organizing and storing data.
- Data Structure is a way of collecting & organizing data in such a way that, we can perform operations on these data in an effective way.
- It also defines the relationship between them.
- **Example**: arrays, Linked list, Stack, Queue etc.

Applications of Data Structure

- Compiler Design
- Operating System
- Database Management System
- Graphics
- Artificial Intelligence
- Numerical analysis

Types of Data Structure

Data structure is mainly classified into two types:

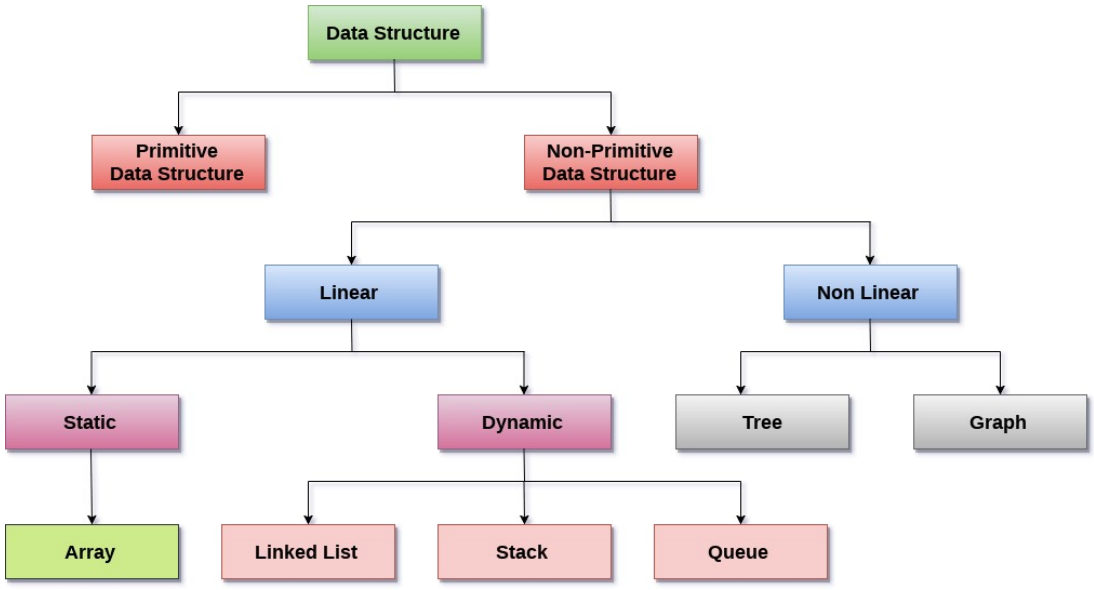
1. Linear Data Structure
2. Non-Linear Data Structure

Linear Data Structure

- A data structure is called linear if all of its elements are arranged in the sequential order.
- **Example**: Linked list, Queue, Array, Stack etc.

Non-Linear Data Structure

- In non-linear data structure, the data elements are not arranged in the sequential structure.
- **Example**: tree, graphs etc.



Algorithm Introduction

- An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Algorithm is not the complete code or program, it is just the **core logic** (solution) of a problem, which can be expressed either as an informal high-level description as **pseudocode** or using a **flowchart**.

- Every algorithm must satisfy the following properties,
  - **Input**: There should be 0 or more inputs supplied externally to the algorithm.
  - **Output**: There should be at least 1 output obtained.
  - **Definiteness**: Every step of the algorithm should be clear and well defined.
  - **Finiteness**: The algorithm should have finite number of steps.
  - **Correctness**: Every step of the algorithm must generate a correct output.

- An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties,
  - **Space complexity**: It is the amount of memory space required by the algorithm, during the course of its execution.
  - **Time complexity**: Time complexity is a way to represent the amount of time required by the program to run till its completion.

Applications of Algorithm

1. Age Group Problems
   - Problems like finding the people of a certain age group can easily be solved with a little modified version of the binary search algorithm.
2. Finding square root of a number
3. Rubik's cube problem
4. DNA Problem
5. Facebook chatbots
6. Stock Market Algorithm
7. Solid Gold Bomb
8. Microsoft's AI Bot
9. Facial Recognition algorithm

Asymptotic Notations

- Asymptotic Notations are the expressions that are used to represent the complexity of an algorithm.
- So, when we want to represent the **growth** of an algorithm, as input increases, we use asymptotic notations.
- The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

- There are **three** types of analysis that we perform on a particular algorithm,
  - **Best Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.
  - **Worst Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.
  - **Average Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

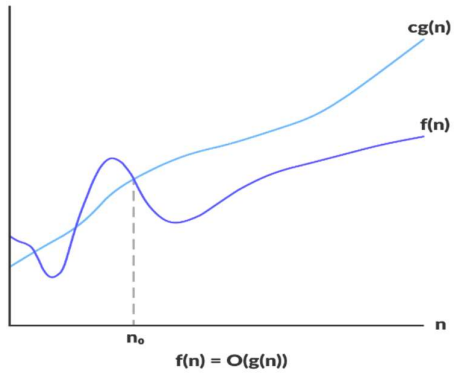
Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of an algorithm, as input increases,

- Big Oh (O) – specifically describes **worst** case scenario
- Omega (Ω) – specifically describes **best** case scenario
- Theta (θ) – represents the **average** complexity of an algorithm

Big – Oh Notation (O)

- Big O notation specifically describes worst case scenario.
- It represents the upper bound running time complexity of an algorithm.
- **Example**:
  - O (1) – Big O notation O (1) represents the complexity of an algorithm that always execute in same time or space regardless of the input data.
  - O (n) – Big O notation O (n) represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data.
  - O (n^2) – Big O notation O (n^2) represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data.
- Similarly, there are other Big O notations such as: logarithmic growth O (log n), exponential growth O (2^n), non-linear growth O (n log n) and factorial growth O (n!).
- Big – O gives the upper bound of a function.



$O(g(n)) = \{f(n)$: there exist positive constants c and n_0
such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

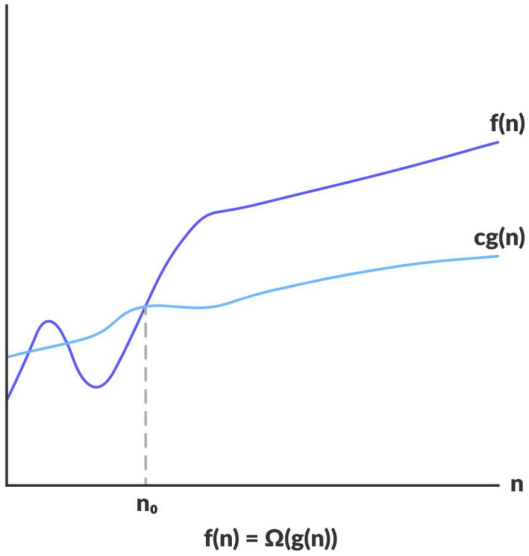
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exist a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyse an algorithm as we are always interested in worst case scenario.

Omega Notation (Ω)

- Omega notation specifically describes best case scenario.
- It represents the lower bound running time complexity of an algorithm.
- So, if we represent a complexity of an algorithm on Omega notation, it means that the **algorithm cannot be completed in less time than this**.



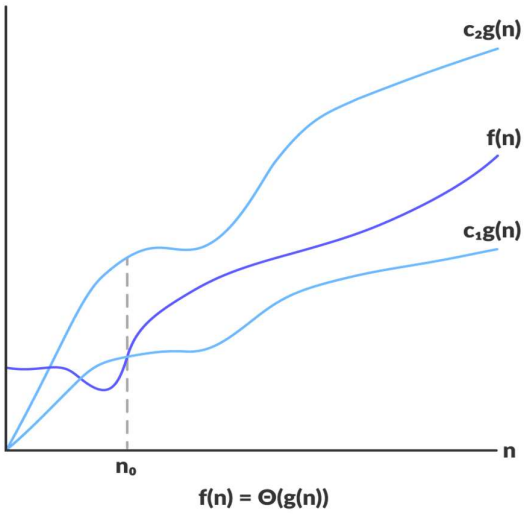
$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0
such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$

the above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

for any value of n, the minimum time required by the algorithm is given by omega Ω(g(n)).

Theta Notation (Θ)

- This notation describes both upper and lower bound of an algorithm so we can say that it defines exact asymptotic behaviour.
- In the real case scenario, the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.
- Theta bounds the functions within constants factors.



f(n) = Θ(g(n))

For a function g(n), Θ(g(n)) is given by the relation,

$$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

The above function can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be lies between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1g(n)$ and $c_2 > g(n)$ for all $n \ge nO$, then f(n) is said to be asymptotically tight bound.

Stack

What is Stack?

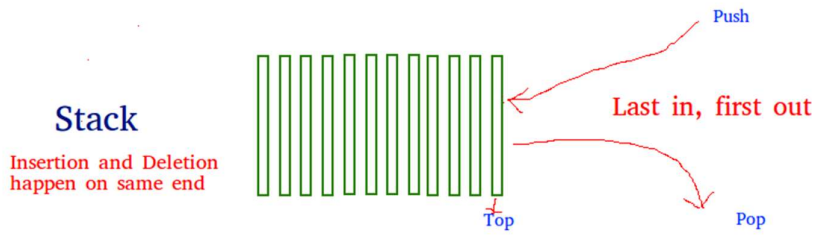
- Stack is a linear data structure which is used to store data in a particular order.
- Two operations that can be performed on stack are:
    - Push operation which inserts an element into the stack.
    - Pop operation which removes the last element that was added into stack.
    - Peek or Top returns top element of stack.
    - isEmpty returns true if the stack is empty, else false.
- It follows last in first out (LIFO) order.

Example

- Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e., the plate which has been placed at the bottommost position remains in the stack for the longest period of time.

Application of stack

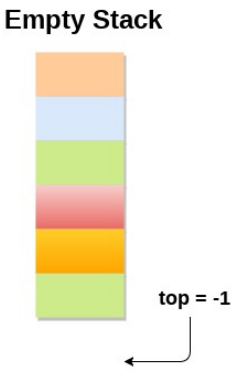
- Recursion
- Expression evaluations and conversions
- Browsers
- Undo in an Editor



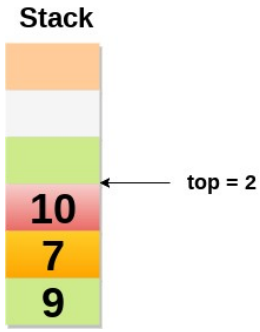
Top position	Status of stack
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

How the Stack grows?

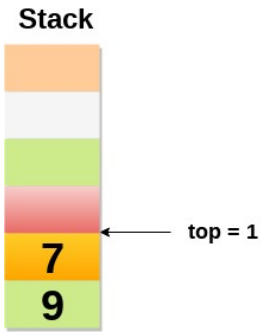
1. Stack is empty
   The stack is called **empty** if it doesn't contain any element inside it. At this stage, the value of variable **top is -1**.



2. Stack is not empty
   Value of top will increase by 1 every time when we add any element to the stack. In the following stack, after adding first element, top = 2.



3. Deletion of an element
   Value of top will get decreased by 1 whenever an element is deleted from the stack.
   In the following stack, after deleting 10 from the stack, **top = 1**.



Steps involved in insertion and deletion of an element in the stack

Push

- Increment the variable top so that it can refer to the next memory allocation.
- Copy the item to the array index value equal to the top.
- Repeat step 1 and 2 until stack overflows.

Pop

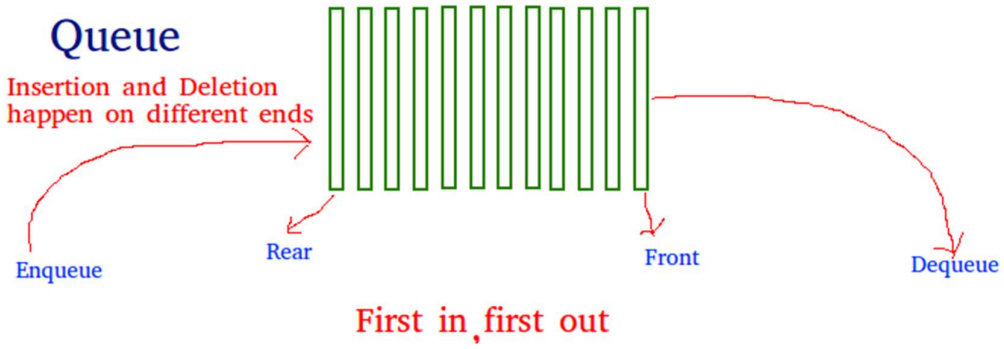
- Store the top most element into another variable
- Decrement the value of the top
- Return the top most element

The difference between stacks and queues is in removing. In stack we remove the most recently added item, but in a queue, we remove last recently added item.

Queue

what is queue?

- A queue is a linear data structure which follows a particular order in which the operations are performed.
- The order is First in First out (FIFO).
- A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stacks and queues is in **removing**. In stack we remove the most recently added item; in a queue, we remove least recently added item.



Applications of Queue

- Queues are widely used as waiting lists for a single shared resource like a printer, disk, CPU.
- Queues are used in the asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for e.g., pipes, file IO, sockets.
- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- Queues are used to maintain the playlist in media players to add and remove the songs from the play-list.
- Queues are used in operating systems for handling interrupts.

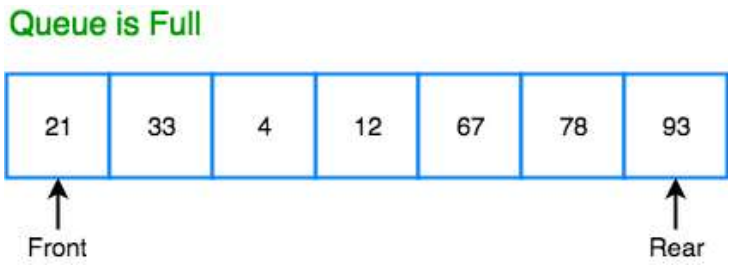
Dequeue

Dequeue (also known as double-ended queue) can be defined as an ordered set of elements in which the insertion and deletion can be performed at both the ends i.e., front and rear.

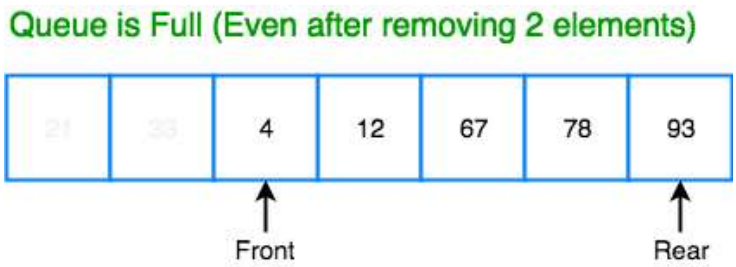
Circular Queue

Why we need a circular queue, when we already have linear queue data structure?

- In a linear queue, once the queue is completely full, it's not possible to insert more elements.
- Even if we dequeue the queue to remove some of the elements, until the queue is reset, no new elements can be inserted.



- When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the **rear** pointer is still at the end of the queue.



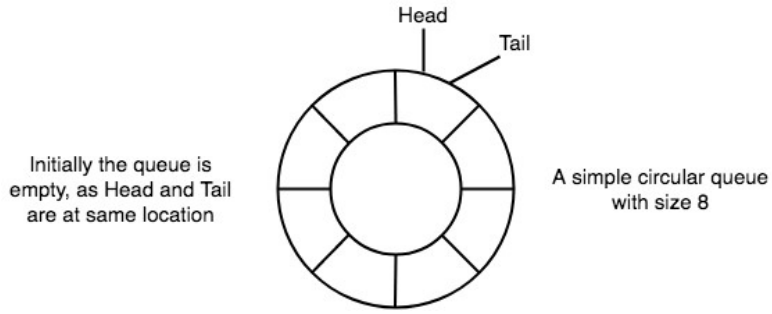
- The only way is to reset the linear queue, for a fresh start.

Circular Queue Definition

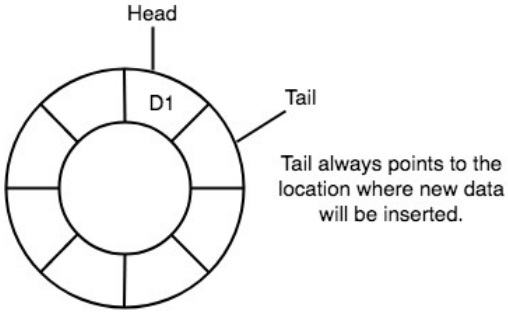
Circular queue is also a linear data structure, which follows the principle of **FIFO** (First in First out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

Queue Operations

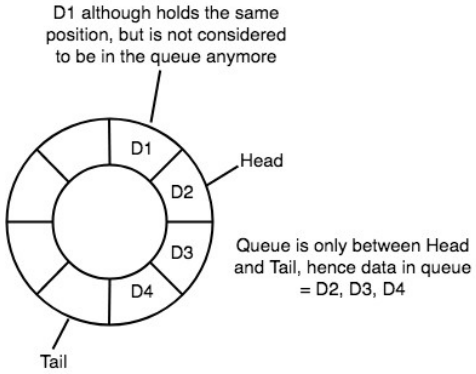
1. In case of a circular queue, **head** pointer will always point to the front of the queue, and **tail** pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is **empty**.



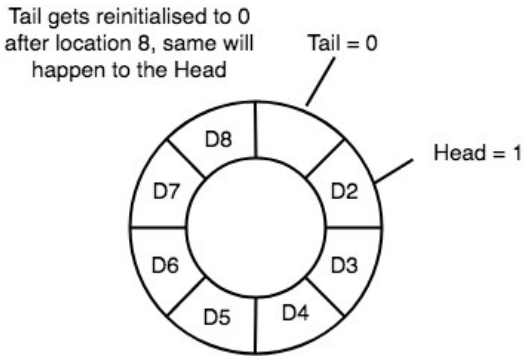
3. New data is always added to the location pointed by the **tail** pointer, and once the data is added, **tail** pointer is incremented to point to the next available location.



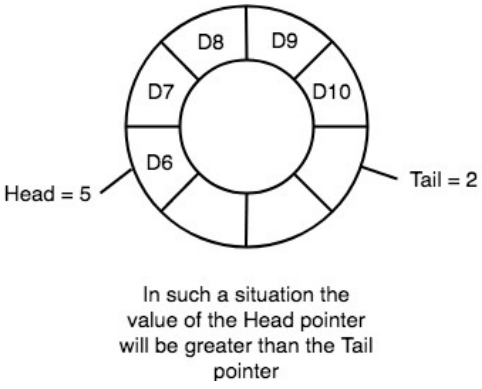
4. In a circular queue, data is not actually removed from the queue. Only the head pointer is increased by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.



5. The head and the tail pointer will get reinitialized to **0** every time they reach the end of the queue.



6. Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialized upon reaching the end of the queue.



Application of Circular Queue

- Computer controlled Traffic Signal System uses circular queue.
- CPU scheduling and memory management.
- Booking Movie Ticket

Implementation of Circular Queue

1. Initialize the queue, with size of the queue defined (**maxSize**), and head and tail pointers.
2. **Enqueue**: check if the number of elements is equal to maxSize – 1
   - If yes, then return Queue is full.
   - If no, then add the new data element to the location of tail pointer and increment the tail pointer.
3. **Dequeue**: check if the number of elements in the queue is zero
   - If yes, then return Queue is empty.
   - If no, then increment the head pointer.
4. Finding the size:
   - If, tail >= head, size = (tail - head) + 1
   - But if, head > tail, then size = maxSize – (head - tail) + 1

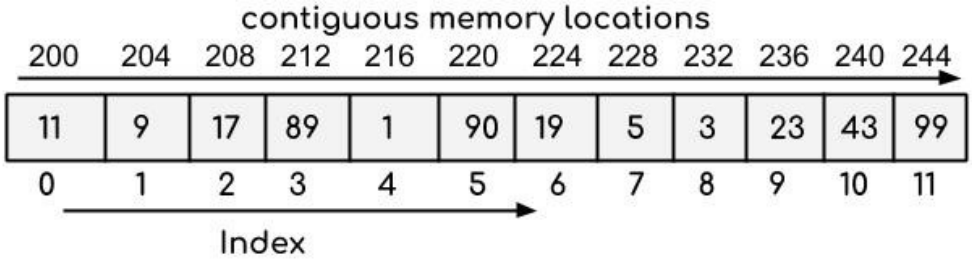
Array

What is an array?

- An array is a collection of homogeneous (same type) data items stored in contiguous memory locations.
- For Example, if an array is of type "int", it can only store integer elements and cannot allow the elements of other types such as double, float, char etc.

Array Representation

The following diagram represents an integer array that has 12 elements. The index of the array starts with 0, so the array having 12 indexes from 0 to 11.



Why we need an array?

- Array is particularly useful when we are dealing with lot of variables of the same type.
- For example, let's say I need to store the marks in math subject of 100 students. To solve this particular problem, either I have to create the 100 variables of int type or create an array of int type with the size 100.
- Obviously the second option is best, because keeping track of all the 100 different variables is a tedious task. On the other hand, dealing with array is simple and easy, all 100 values can be stored in the same array at different indexes (0 to 99).
- Arrays are best for storing multiple values in a single variable.
- Arrays are better at processing many values easily and quickly.
- Sorting and searching the values is easier in arrays.

Array Operations

Insert

With this operation, you can insert one or more items into an array at the beginning, end, or any given index of the array.

Delete

With this operation, you can delete one item from an array by value. This method accepts only one argument, value. After running this method, the array items are re-arranged, and indices are reassigned.

Search

With this operation, you can search for an item in an array based on its value. This is a non-destructive method, which means it does not affect the array values.

Update

This operation is quite similar to the insert method, except that it will replace the existing value at the given index. This means will simply assign a new value at the given index.

Traverse

This method prints all the array element one by one. This operation is to traverse through the elements of an array.

Applications of array

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables.
- Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques and stacks.

Real-Life Examples

1. Post office boxes
2. Book pages
3. Egg cartons
4. Chess/checkerboards

Advantages of array

- It is better and convenient way of storing the data of same datatype with same size.
- It allows us to store known number of elements in it.
- It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/memory for its elements. Hence there is no memory overflow or shortage of memory in array.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc.
- It allows to store the elements in any dimensional array – supports multidimensional array.

Disadvantages

- It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. Hence if we need to insert more no of records than declared then it is not possible.
- Inserting and deleting the records from the array would be costly since we add/delete the elements from the array. We need to manage memory space too.
- It does not verify the indexes while compiling the array. In case there is any indexes pointed which is more than the dimension specified, then we will get run time errors rather than identifying them at compile time.

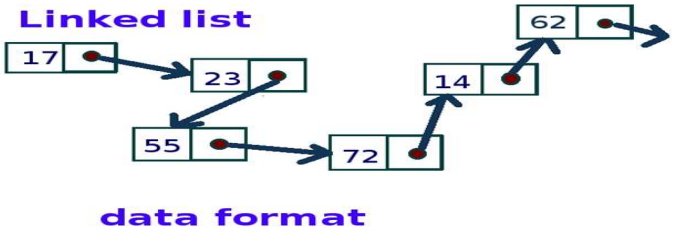
Multi-dimensional array

The multi-dimensional array can be defined as the array of arrays in which, the data is stored in tabular form consists of rows and columns. 2D arrays are created to implement a relational database lookalike data structure.

Linked List

Define LinkedList in Data Structure?

- Linked list is the collection of randomly stored data objects called **nodes**. In linked list, each node is linked to its adjacent node through a pointer.
- A node contains two fields, i.e., **Data** field and **link** field.



- A linked list is considered as both linear and non-linear data structure depending upon the situation,
  - On the basis of data **storage**, it is considered as a non-linear data structure.
  - On the basis of the **access strategy**, it is considered as a linear data structure.

What are the advantages of linked list over an array?

- The size of a linked list can be incremented at runtime which is impossible in the case of the array.
- The list is not required to be contiguously present in the main memory, if the contiguous space is not available, the nodes can be stored anywhere in the memory connected through the links.
- The list is dynamically stored in the main memory and grows as per the program demand while the array is statically stored in the main memory, size of which must be declared at compile time.
- The number of elements in the linked list are limited to the available memory space while the number of elements in the array is limited to the size of an array.

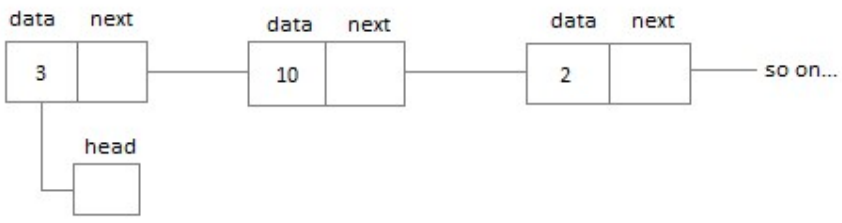
Types of Linked list

There are 3 different implementations of linked list available, they are,

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

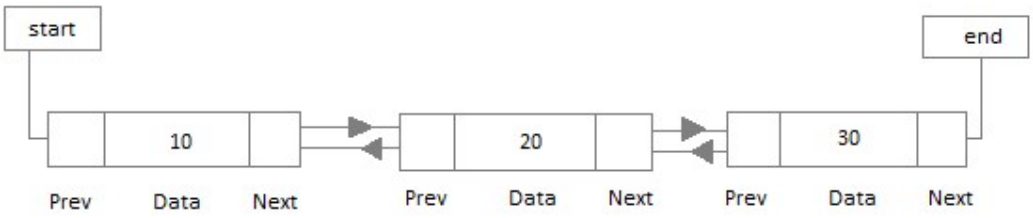
Singly Linked List

- Singly linked lists contain nodes which have a **data** part as well as an **address** part i.e., next, which points to the next node in the sequence of nodes.
- The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



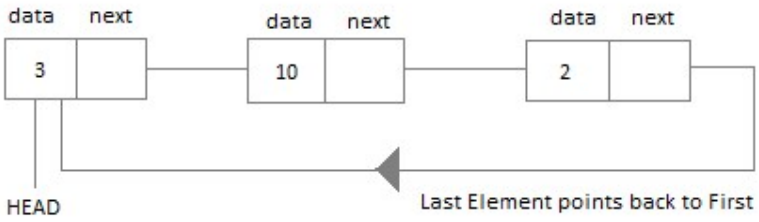
Doubly Linked List

In a doubly linked list, each node contains a **data** part and **two address** part, one for the **previous** node and one for the **next** node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



Operations on Linked List

Traversal

- To traverse all the nodes one after another.
- This operation is to step through the list from beginning to end.
- **Example**: we may want to print the list or search for a specific node in the list.
- The algorithm for traversing a list,
  - Start with the head of the list. Access the content of the head node if it is not null.
  - Then go to the next node (if exists) and access the node information.
  - Continue until no more nodes (i.e., you have reached the null node)

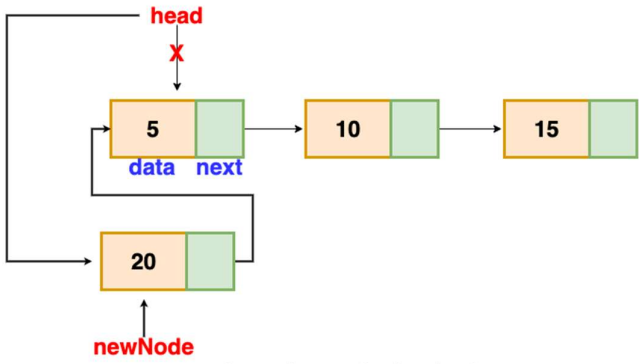
Insertion

There are three cases of inserting a node in a linked list,

- Insertion at the beginning
- Insertion at the end
- Insertion after a given specified node

Insertion at the beginning

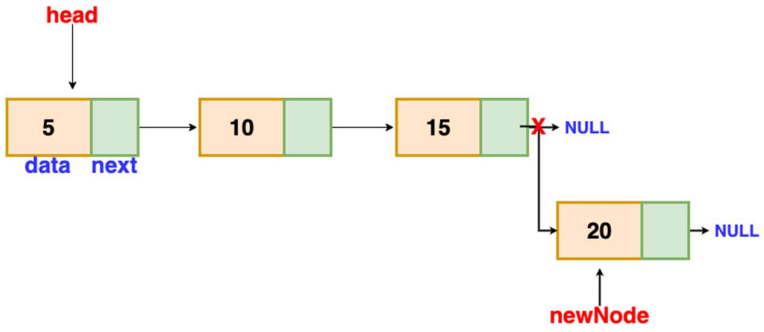
Since there is no need to find the end of the list. If the list is empty, we make the new node as the head of the list. Otherwise, we have to connect the new node to the current head of the list and make the new node, the head of the list.



Insertion at the beginning

Insertion at the end

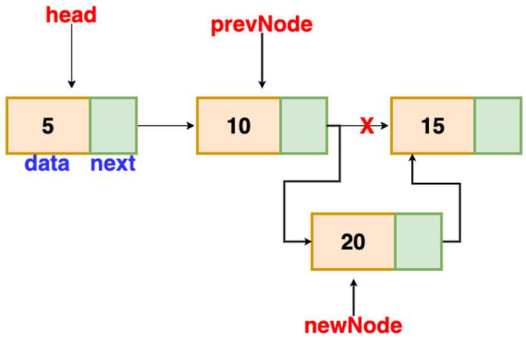
- We will traverse the list until we find the last node. Then we insert the new node to the end of the list. Note that we have to consider special cases such as list being empty.
- In case of a list being empty, we will return the updated head of the linked list because in this case, the inserted node is the first as well as the last node of the linked list.



Insertion at the end

Insertion after a given specified node

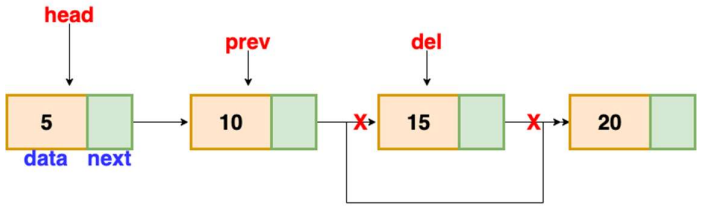
- We are given the reference to a node, and the new node is inserted after the given node.
- If the address of the previous node is not given, then you can traverse to that node by finding the data value.



Insertion after a given node

Deletion

- To delete a node from a linked list, we need to do these steps,
    1. Find the previous node of the node to be deleted.
    2. Change the next pointer of the previous node.
    3. Free the memory of the deleted node
- In the deletion, there is a special case in which the first node is deleted. In this, we need to update the head of the linked list.



Deleting a Node in Linked List

- There are 3 cases of deleting a node from the linked list,
    - **Deletion at beginning:**
      It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just needs a few adjustments in the node.
    - **Deletion at the end:**
      It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
    - **Deletion after specified node**
      It involves deleting the node after the specified node in the list. We need to skip the desired number of nodes after which the node will be deleted. This requires traversing through the list.

Searching

- To search any value in the linked list, we can traverse the linked list and compares the value present in the node.
- In searching, we match each element of the list with the given element. If the element is found on any of the location, then location of that element is returned otherwise null is returned.

Update

To update the value of the node, we just need to set the data part to the new value.

Applications of Linked List

- Implementation of stack and queues
- Implementation of graphs
- Dynamic memory allocation
- Performing arithmetic operations on long integers

Advantages

- Linked list is dynamic data structure
- Linked list can grow and shrink during run time
- Insertion and deletion operations are easier
- Efficient memory utilization i.e.no need to pre-allocate memory
- Faster access time

Disadvantages

- They use more memory than arrays because of the storage used by their pointers.
- Difficulties arise in linked list when it comes to reverse traversing.
- Time consuming
- No random access

Hashing

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as Hashing Algorithm or Message Digest Function.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O (1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

Hash Function

- A fixed process converts a key to a hash key is known as Hash Function.
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash**.
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

Hash Table

- Hash table is a data structure that represents data in the form of key-value pairs.
- Each key is mapped to a value in hash table.
- The keys are used for indexing the values/data.
- Hash table is synchronized and contains only unique elements.

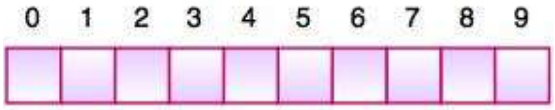


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- Suppose we have integer items {26, 70, 18, 31, 54, 93}.
  One common method of determining a hash key is the **division method of hashing** and the formula is:

  Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

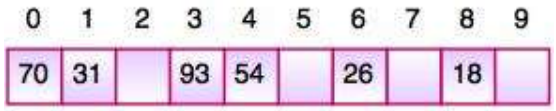


Fig. Hash Table

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, **λ = No. of items / table size**. For example, λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.

Linear Probing

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- It is a component of open addressing scheme for using a hash table to solve the dictionary problem.
- The simplest method is called Linear Probing. Formula to compute linear probing is:

P = (1 + P) % (MOD) Table size

Example:

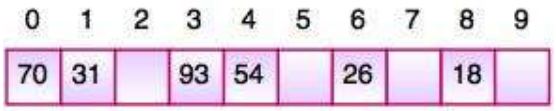


Fig. Hash Table

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

P = H (40)

44 % 10 = **0**

Position 0 is occupied by 70. so, we look elsewhere for a position to store 40.

Using Linear Probing:

P= (P + 1) % table-size

0 + 1 % 10 = **1**

But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position: 1 + 1 % 10 = 2

Position 2 is empty, so 40 is inserted there.

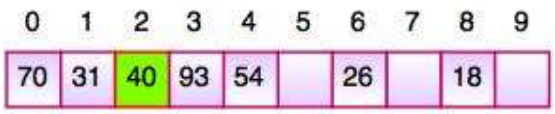


Fig. Hash Table

Applications of hashing

- Message digest
- Password verification
- Data Structures
- Compiler operations
- Linking file name and path together

Advantages

- Hashing provides a more reliable and flexible method of data retrieval than any other data structure.
- It is faster than searching arrays and lists.
- Main advantage is synchronization.
- In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches and sets.

Disadvantages

- Hash table become quite inefficient when there are many collisions.
- Hash table does not allow null values, like hash map.

Tree

What is a tree in data structure?

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked list, stack and queue.
- It represents the nodes connected by edges.

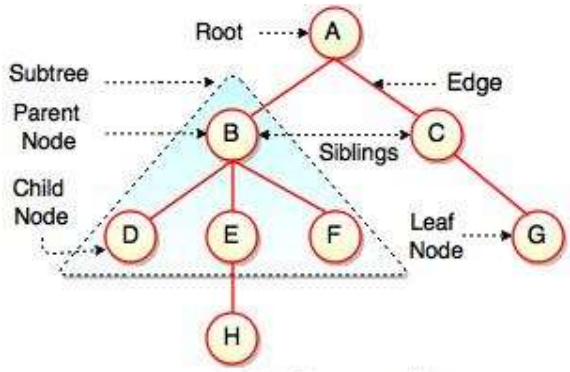


Fig. Structure of Tree

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

Binary Tree

- A tree in which every node can have maximum of two children is called binary tree.
- In binary tree, every node can have a maximum of 2 children, which are known as **Left Child** & **Right Child**.
- It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM).

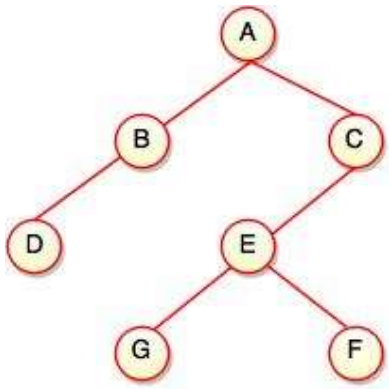


Fig. Binary Tree

Binary Search Tree

- Binary search tree is an ordered binary tree.
- All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element.
- Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.
- Binary search trees are used in most of the applications of computer science domain like searching, sorting etc.
- There are many operations which can be performed on a BST,
    - Searching
    - Insertion
    - Deletion

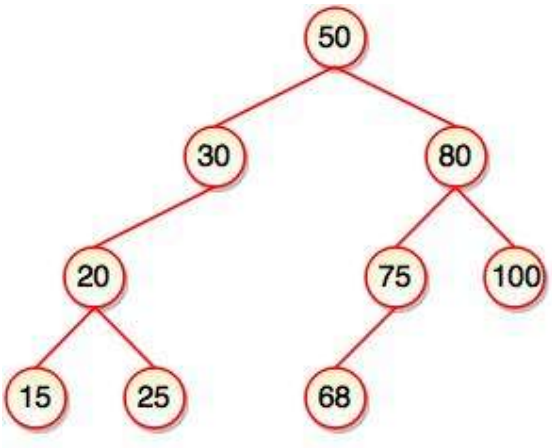


Fig. Binary Search Tree

Binary Tree Traversal

- Binary tree traversing is a process of accessing every node of the tree and exactly once.
- There are techniques of traversal
  - Preorder traversal
  - Postorder traversal
  - Inorder traversal

Preorder Traversal

Algorithm for preorder traversal

1. Start from the root
2. Then, go to the left subtree
3. Then, go to the right subtree

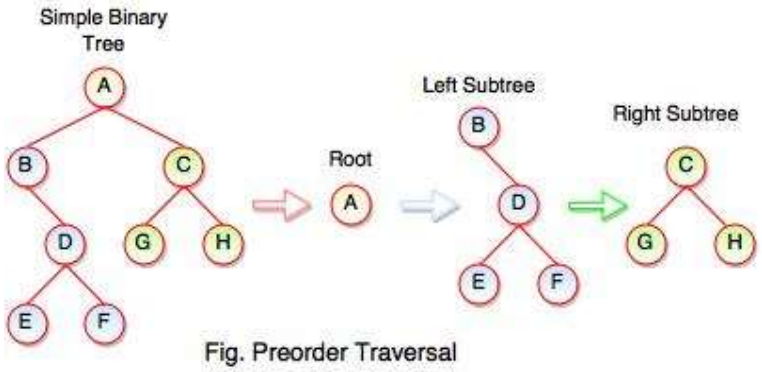


Fig. Preorder Traversal

Preorder Traversal: A B C D E F G H

Postorder Traversal

Algorithm for Postorder traversal

1. Start from the left subtree (last leaf)
2. Then, go to the right subtree
3. Then, go to the root

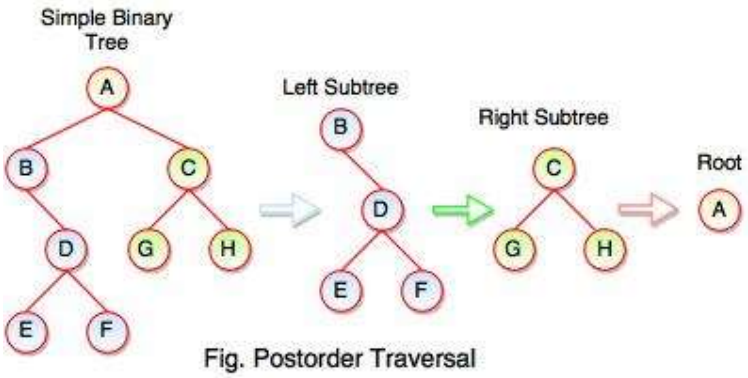


Fig. Postorder Traversal

Postorder Traversal: E F D B G H C A

Inorder Traversal

Algorithm for Inorder traversal

1. Start from the left subtree
2. Then, visit the root
3. Then, go to the right subtree

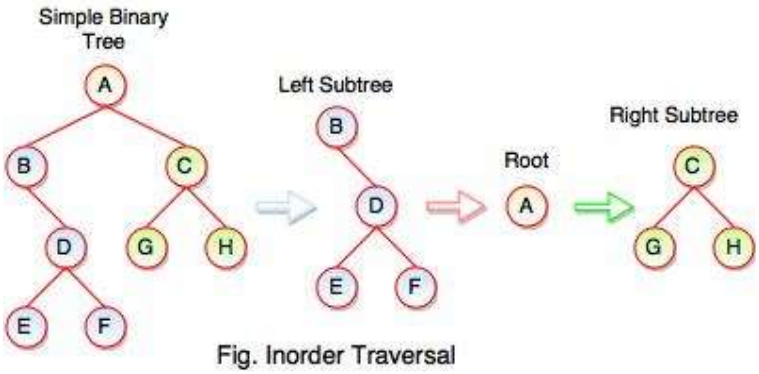


Fig. Inorder Traversal

Inorder Traversal: B E D F A G C H

Applications of tree

- Represent organization
- Represent computer file systems
- Networks to find best path in the internet
- Chemical formula representation
- Binary search tree
- Decision tree

Advantages

- Tree reflects structural relationship in the data
- It is used to represent hierarchies
- It provides an efficient insertion and searching operations.
- Trees are flexible. It allows to move subtree around with minimum effort.

Disadvantages

- Binary tree is only efficient when the tree is balanced. If the tree is unbalanced, it won't provide same benefits.
- Data storage on the heap is slower and comparison takes more time than other data structure.

Heap

A heap is a special tree-based data structure in which the tree is a complete binary tree.

Generally, heaps can be of two types

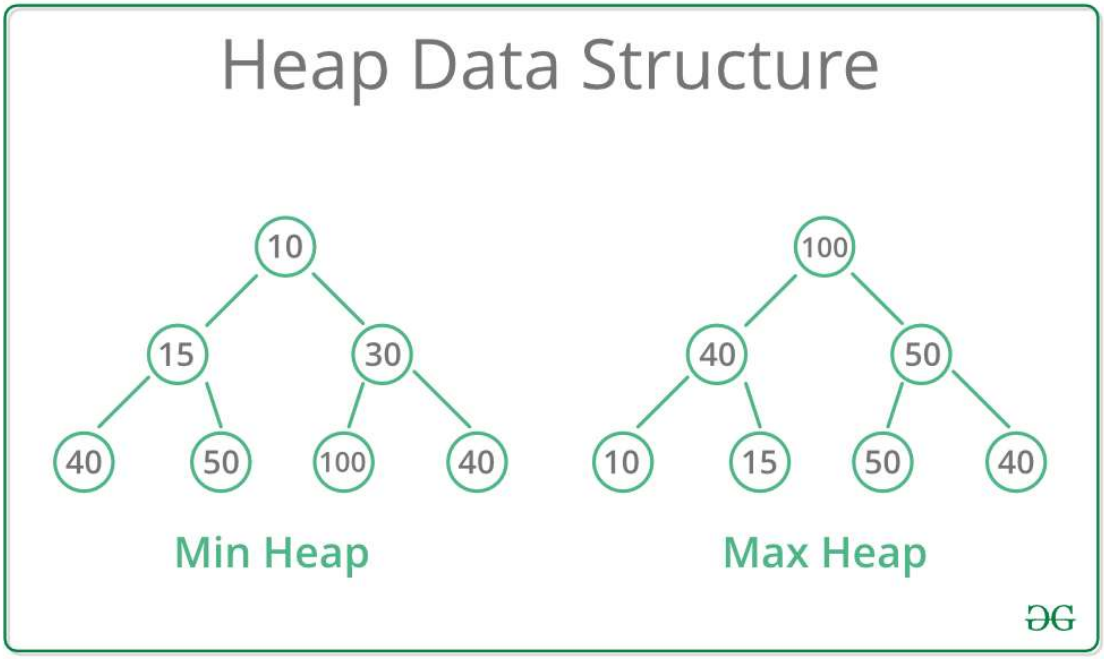
Max-Heap

In a max-heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub trees in that binary tree.

Min-Heap

In min-heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that binary tree.

Example



Applications of Heap Data Structure

- Heap sort
- Graph algorithm like Dijkstra's algorithm
- Priority queue
- But the job it does is optimized the algorithms by providing a means for efficient min-max of lists and arrays.

Graphs

- Graph is an abstract data type.
- It is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- Graph is used to implement the undirected graph and directed graph concepts from mathematics.
- It represents many real-life applications. Graphs are used to represent the networks. Network includes path in a city, telephone network etc.
- It is used in social networks like Facebook, LinkedIn etc.

Components of Graph

- Graph consists of two following components,
    - Vertices
    - Edges
- Graph is a set of vertices (V) and set of edges (E).
- V is a finite number of vertices also called as **nodes**.
- E is a set of ordered pair of vertices representing **edges**.
- For **example**, in Facebook, each person is represented with a vertex or a node. Each node is a structure and contains the information like user id, user name, gender etc.

Example

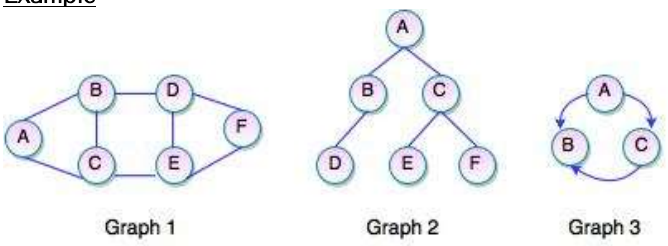


Fig. Graphs

Graph 1:
V = {A, B, C, D, E, F}
E = {(A, B), (A, C), (B, C), (B, D), (D, E), (D, F), (E, F)}

Graph 2:
V = {A, B, C, D, E, F}
E = {(A, B), (A, C), (B, D), (C, E), (C, F)}

Graph 3:
V = {A, B, C}
E = {(A, B), (A, C), (C, B)}

Directed Graph

- If a graph contains ordered pair of vertices, is said to be a Directed Graph.
- If an edge is represented using a pair of vertices (V_1, V_2), the edge is said to be directed from V_1 to V_2.
- The first element of the pair V_1 is called the start vertex and the second element of the pair V_2 is called the end vertex.

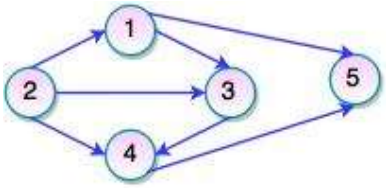


Fig. Directed Graph

Set of Vertices V = {1, 2, 3, 4, 5, 5}

Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

Undirected Graph

- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.

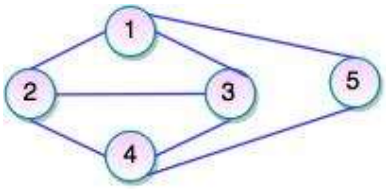


Fig. Undirected Graph

Set of Vertices V = {1, 2, 3, 4, 5}
Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

- In an undirected graph, the nodes are connected by undirected arcs.
- It is an edge that has no arrow. Both the ends of an undirected arc are equivalent, there is no head or tail.

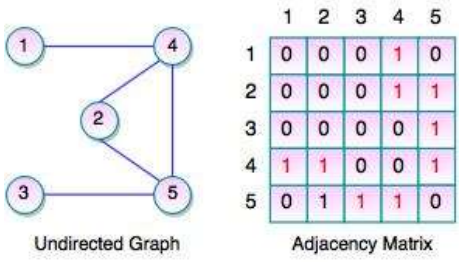
Applications of Graph

- In computer science graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate th shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to another page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind google page ranking algorithm.

Graph Representation

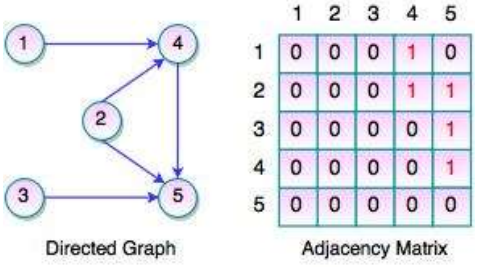
Adjacency Matrix

- Adjacency matrix is a way to represent a graph.
- It shows which nodes are adjacent to one another.
- Graph is represented using a square matrix.
- Adjacency matrix of an undirected graph is always a symmetric matrix which means an edge (i, j) implies the edge (j, i).
- Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicated a directed edge from vertex i to vertex j.



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Fig. Adjacency Matrix Representation of Undirected Graph



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

Fig. Adjacency Matrix Representation of Directed Graph

Master Theorem

Master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.

The master method is a formula for solving recurrence relations of the form:

$T(n) = aT(n/b) + f(n)$,

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed

to have the same size.

f(n) = cost of the work done outside the recursive call,

which includes the cost of dividing the problem and

cost of merging the solutions

Here, $a \geq 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function.

If $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function, then the time complexity of a recursive relation is given by,

$T(n) = aT(n/b) + f(n)$

where, T(n) has the following asymptotic bounds:

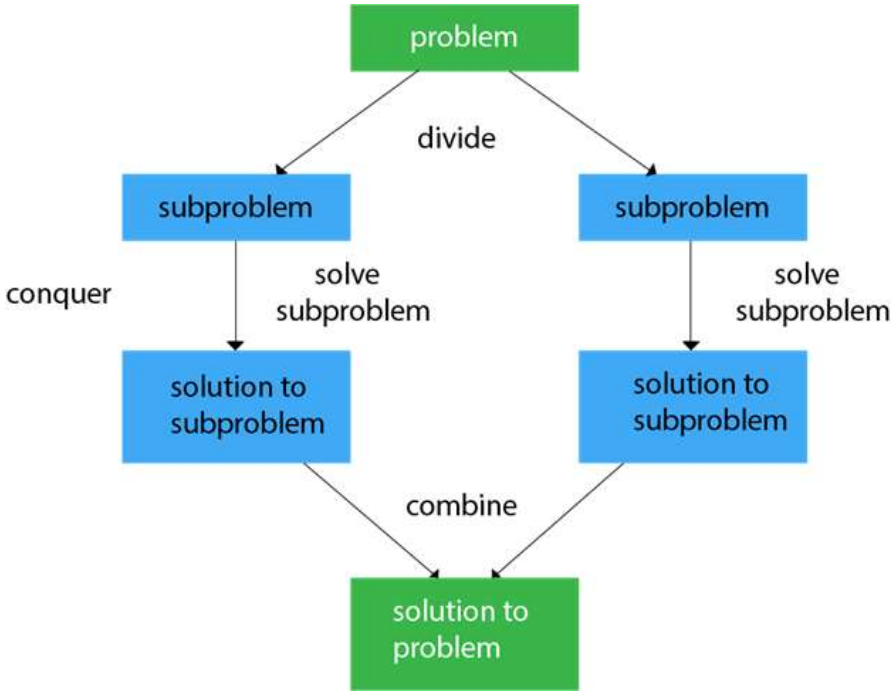
1. If $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

$\epsilon > 0$ is a constant.

Divide and Conquer Algorithm

The steps involved are,

1. **Divide**
   Divide the given problem into sub-problems using recursion.
2. **Conquer**
   Solve the smaller sub-problems recursively. If the sub-problem is small enough, then solve it directly.
3. **Combine**
   Combine the solutions of the sub-problems which is part of the recursive process to get the solution to the actual problem.



Examples:

The specific computer algorithms are based on Divide & Conquer approach,

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (Merge & Quick Sort)
4. Tower of Hanoi

Recursion

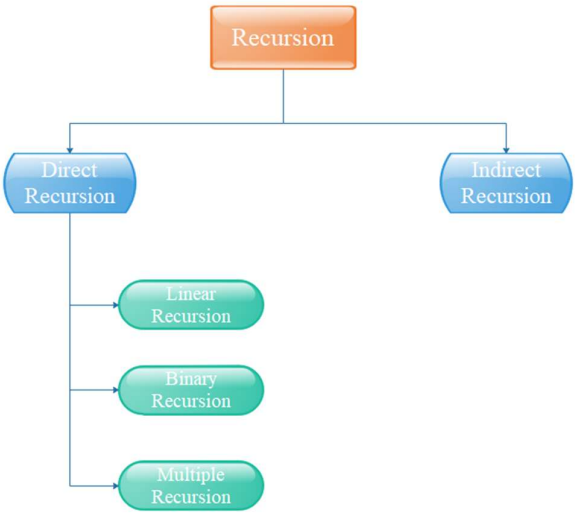
The process in which a function calls itself directly or indirectly is called **recursion** and corresponding function is called as **recursive function**.

Using recursive algorithm, certain problems can be solved quite easily. Like,

1. Towers of Hanoi
2. Inorder/Preorder/Postorder Tree Traversal
3. DFS of Graph

Example:

The Fibonacci Sequence is defined as: F(i) = F(i-1) + F(i-2)



Disadvantages of Recursion

- It consumes more storage space as the recursive calls along with automatic variables are stored on the stack.
- The computer may run out of memory if the recursive calls are not checked.
- It is not more efficient in terms of speed and execution time.
- Recursive solution is always logical and it is very difficult to trace.
- Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.

Searching Algorithm

Searching

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.

Searching Techniques

To search an element in a given array, it can be done in following ways,

1. Sequential Search/Linear Search
2. Binary Search

Sequential/Linear Search

- Sequential search is also called as Linear Search.
- It is a basic and simple search algorithm.
- Sequential search starts at the beginning of the list and checks every element of the list.
- Sequential search compares the elements with all other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

Code Logic

```
unction searchValue (value, target)
{
    for (var i = 0; i < value.Length; i++)
    {
        if (value[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Example

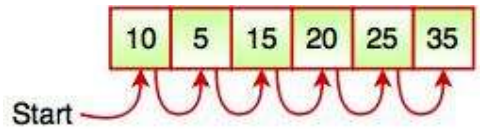


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element **25**, it will go step by step in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

Binary Search

- Binary search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O (log n).
- Binary search works on the principle of divide & conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Code Logic

```
do until the pointers low and high meet each other.
   mid = (low + high)/2
   if (x == arr[mid])
      return mid
   else if (x > A[mid]) // x is on the right side
      low = mid + 1
   else                 // x is on the left side
      high = mid - 1
```

Example:

1. The array in which searching is to be performed is:

   Let x= 4 be the element to be searched.
2. Set two pointers low and high at the lowest and the highest positions respectively.

3. Find the middle element mid of the array i.e. (arr [low + high]) / 2 = 6.

4. If x == mid, then return mid. Else, compare the element to be searched with m.
5. If x > mid, compare x with the middle element of the elements on the right side of mid. This is done by setting low to low = mid + 1.
6. Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to high = mid - 1.

7. Repeat steps 3 to 6 until low meets high.

8. X = 4 is found.

Sorting Algorithm

Sorting

- Sorting is a process of arranging item in ascending or descending order.
- It arranges the data in a sequence which makes searching easier.

Example:

Suppose we have a record of employee. It has the following data,

Employee No.

Employee Name

Employee Salary

Department Name

Here, employee no. can be takes as key for sorting the records in ascending or descending order. Now, we have to search an Employee with employee no. 116, so we don't require to search the complete record, simply we can search between the Employees with employee no. 100 to 120.

Sorting techniques

Sorting technique depends on the situation. It depends on two parameters,

- Execution time of program that means time taken for execution of program.
- Space that means space taken by the program.

Sorting can be performed using below listed techniques,

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Counting Sort
7. Radix Sort
8. Bucket Sort
9. Heap Sort
10. Shell Sort

Bubble Sort

- This algorithm is based on the idea of repeatedly comparing pairs of adjacent elements and then switching their positions if they exist in the wrong order.
- Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order.
- The order can be ascending or descending.

Code Logic

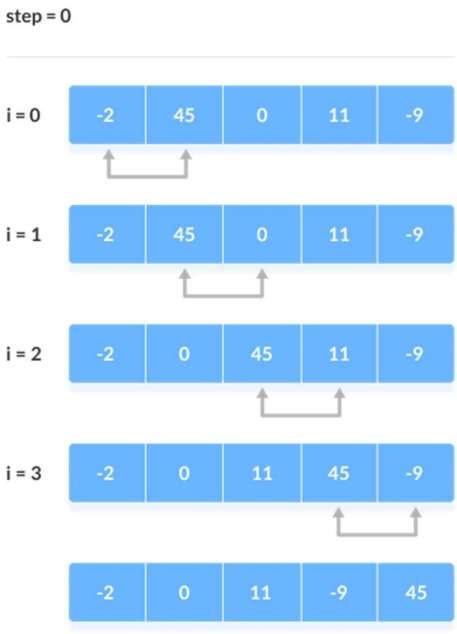
```
bubbleSort(array)
 for i <- 1 to indexOfLastUnsortedElement-1
  if leftElement > rightElement
   swap leftElement and rightElement
end bubbleSort
```

Example

1. Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped.

   Now, compare the second and the third elements. Swap them if they are not in order.

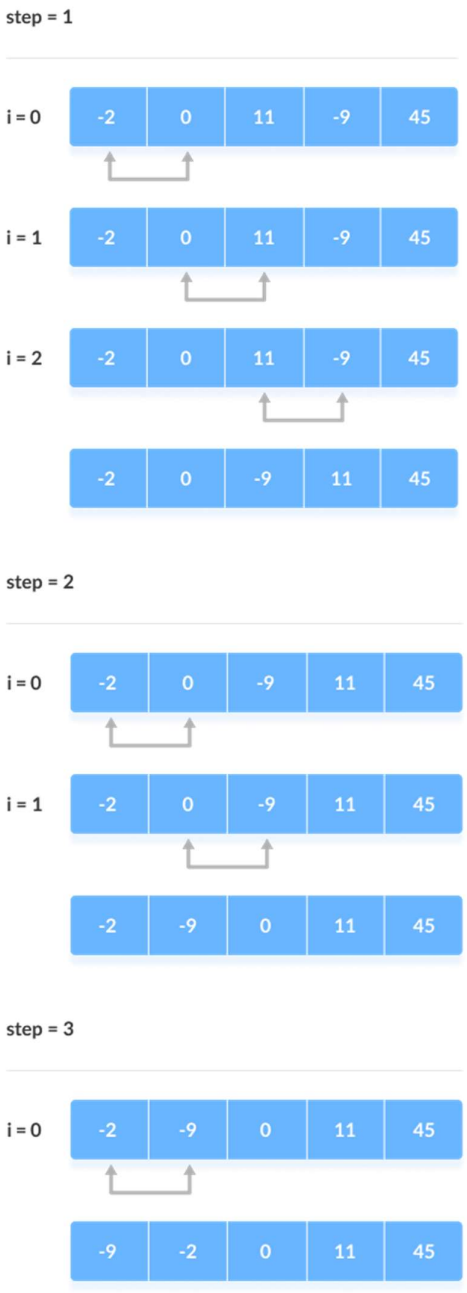
   The above process goes on until the last element.



2. The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.

In each iteration, the comparison takes place up to the last unsorted element.

The array is sorted when all the unsorted elements are placed at their correct positions.



Bubble Sort Complexity

Complexity – O (n^2)

Time Complexity

- **Worst Case Complexity: O (n^2)**
  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

- **Best Case Complexity: O (n)**
  If the array is already sorted, then there is no need for sorting.

- **Average Case Complexity: O (n^2)**
  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

Space Complexity

- Space complexity is **O (1)** because an extra variable temp is used for swapping.

Applications

Bubble sort is used in the following cases where,

1. The complexity of the code does not matter.
2. A short code is preferred.

- Bubble sort is a sorting algorithm that is used to sort the elements in an ascending order.
- It uses less storage space.
- Bubble sort can be beneficial to sort the unsorted elements in a specific order.
- It can be used to sort the students on basis of their height in a line.

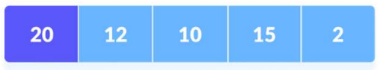
Selection Sort

- Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.
- Generally, this algorithm is based on the idea of finding the minimum or maximum element in the unsorted array and then putting in its correct position on a sorted array.
- The complexity is **O (n^2)**.

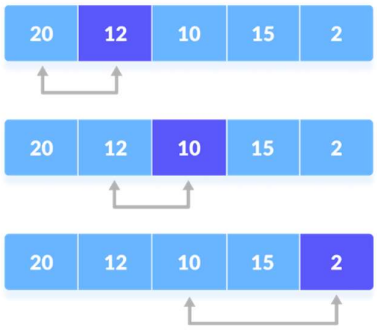
```
selectionSort (array, size) // Code Logic
 repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
   if element < currentMinimum
    set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

Example

1. Set the first elements as minimum.

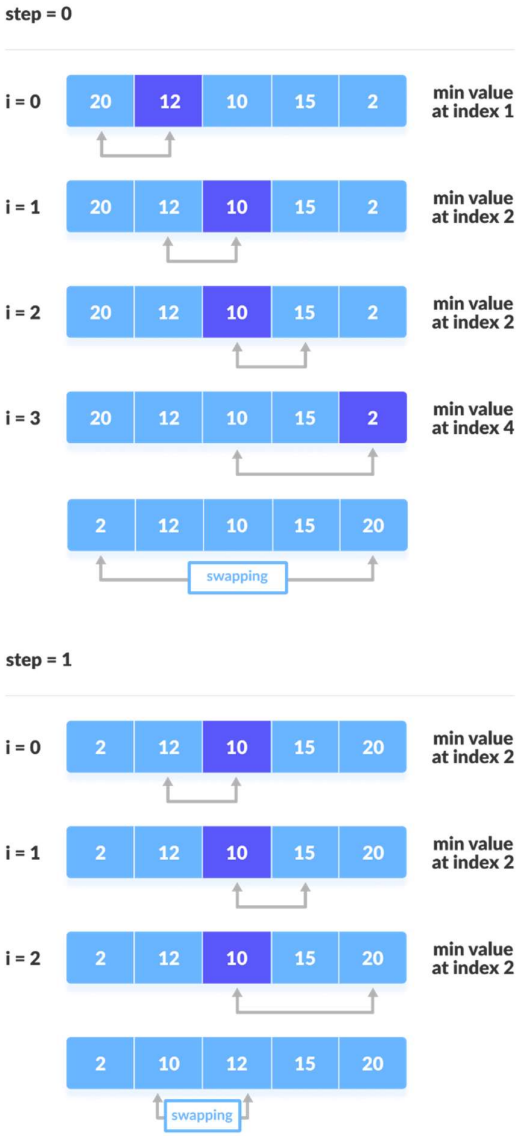


2. Compare **minimum** with the second element. If the second element is smaller than **minimum**, assign the second element as **minimum**.
   Compare **minimum** with the third element. Again, if the third element is smaller, then assign **minimum** to the third element otherwise do nothing. The process goes in until the last element.



3. After each iteration, minimum is placed in the front of the unsorted list.

4. For reach iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



Applications of Selection Sort

The selection sort is used when,

- A small list is to be sorted
- Cost of swapping does not matter
- Checking of all the elements is compulsory

Insertion Sort

- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- The idea behind the algorithm is that, in each iteration, it consumes one element from the input elements, removes it and finds its correct position i.e., where it belongs in the sorted list and places it there.
- Insertion sort works similarly a we sort cards in our hand in a card game.

  We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.

Insertion sort algorithm

```
insertionSort(array)
 mark first element as sorted
 for each unsorted element X
  'extract' the element X
  for j <- lastSortedIndex down to 0
   if current element j > X
    move sorted element to the right by 1
  break loop and insert X here
end insertionSort
```

Complexity

- The complexity of Insertion Sort is O (n^2).
- Worst case time complexity: O (n^2)
- Best case time complexity: O (n)
- Average case time complexity: O (n^2)
- Space complexity: O (1)

Insertion sort application:

The insertion sort is used when,

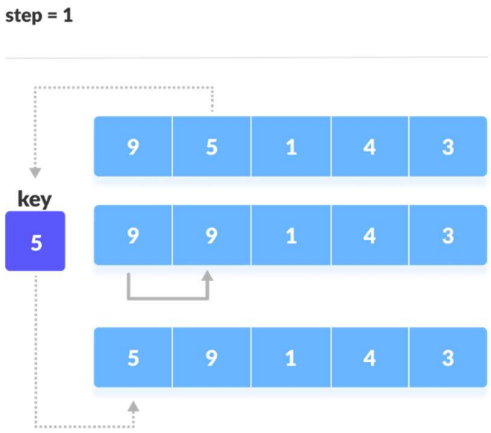
- The array has a small number of elements
- There are only a few elements left to be sorted

Example

Suppose we need to sort the following array,

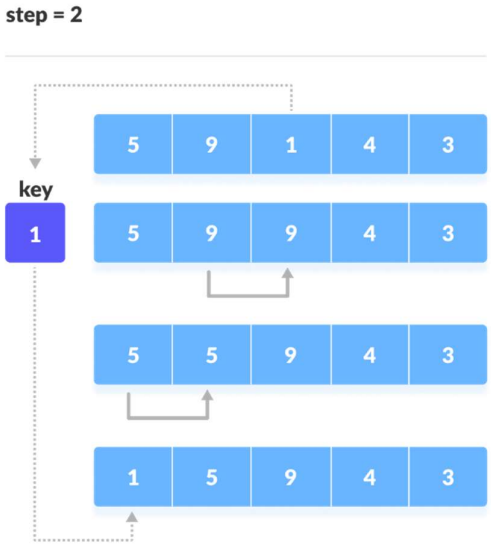
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in **key**.

   Compare **key** with the first element. If the first element is greater than **key**, then **key** is placed in front of the first element.

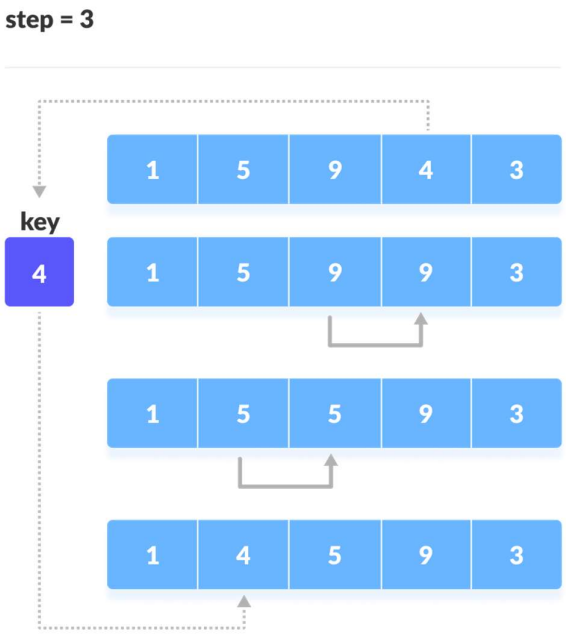


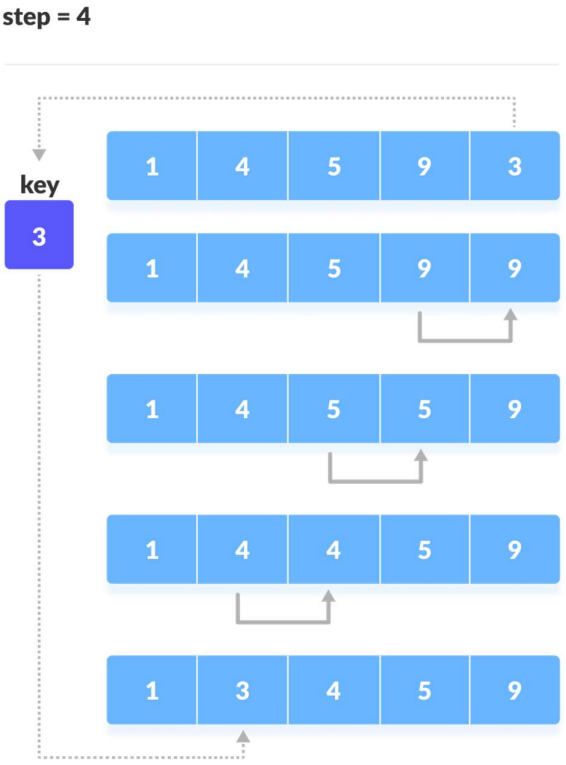
2. Now, the first two elements are sorted.

   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



3. Similarly, place every unsorted element at its correct position.





Merge Sort

- This sorting algorithm works on the principle that – Divide the array into two halves. Repeatedly sort each half, then merge two halves.
- Merge sort is a kind of Divide & Conquer algorithm in computer programming.
- It is one of the most popular sorting algorithms and great way to develop confidence in building recursive algorithms.

Merge sort algorithm

```
MergeSort (A, p, r):
  if p > r
     return
  q = (p+r)/2
  mergeSort (A, p, q)
  mergeSort (A, q+1, r)
  merge (A, p, q, r)
```

Merge Sort Complexity

Time Complexity

Best Case Complexity: **O (n*log n)**

Worst Case Complexity: **O (n*log n)**

Average Case Complexity: **O (n*log n)**

Space Complexity

The space complexity of merge sort is **O (n)**.

Applications

- Inversion count problem
- External sorting
- E-commerce applications

Example

The steps involved are,

Suppose we had to sort an array **A**. A subproblem would be to sort a sub-section of this array starting at index **p** and ending index **r**, denoted as **A [p .... r]**.

Divide

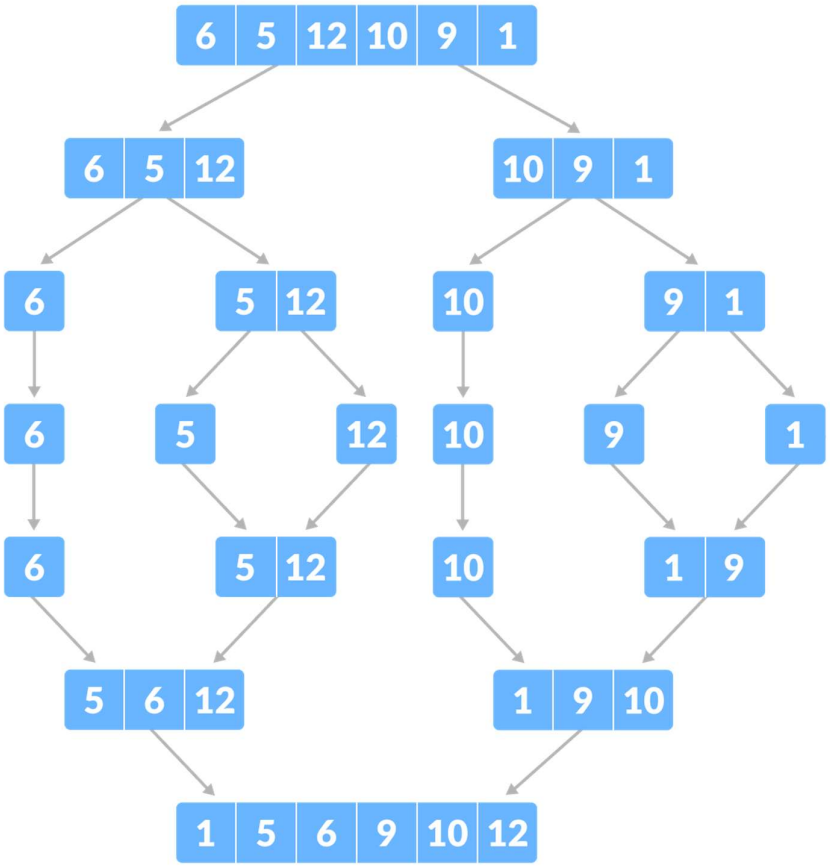
If **q** is the half-way point between **p** and **r**, then we can split the subarray **A [p ... r]** into two arrays **A [p...q]** and **A [q+1 ... r]**.

Conquer

In the conquer step, we try to sort both the subarrays **A [p ... q]** and **A [q+1 ... r]**. if we haven't reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A [p ... q] and A [q+1 ... r] for array A [p ... r], we combine the result by creating a sorted array A [p ... r] from two sorted subarrays A [p ... q] and A [q+1 ... r].



Quick Sort

- This algorithm is also based on the divide and conquer approach. It reduces the space complexity and removes the use of auxiliary array used in merge sort.
- It is based on the idea of choosing one element as **pivot** element and partitioning the array around it such that the left side of pivot contains all elements less than the pivot element and right side contains all elements greater than the pivot.

Complexity

- Worst case time complexity: O (n^2)
- Best case time complexity: O (n*log n)
- Average case time complexity: O (n*log n)
- Space Complexity: O (log n)

Quick shot algorithm

```
quickSort (array, leftmostIndex, rightmostIndex)
 if (leftmostIndex < rightmostIndex)
   pivotIndex <- partition (array, leftmostIndex, rightmostIndex)
   quickSort (array, leftmostIndex, pivotIndex)
   quickSort (array, pivotIndex + 1, rightmostIndex)


partition (array, leftmostIndex, rightmostIndex)
 set rightmostIndex as pivotIndex
 storeIndex <- leftmostIndex - 1
 for i <- leftmostIndex + 1 to rightmostIndex
 if element[i] < pivotElement
   swap element[i] and element[storeIndex]
   storeIndex++
 swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Example

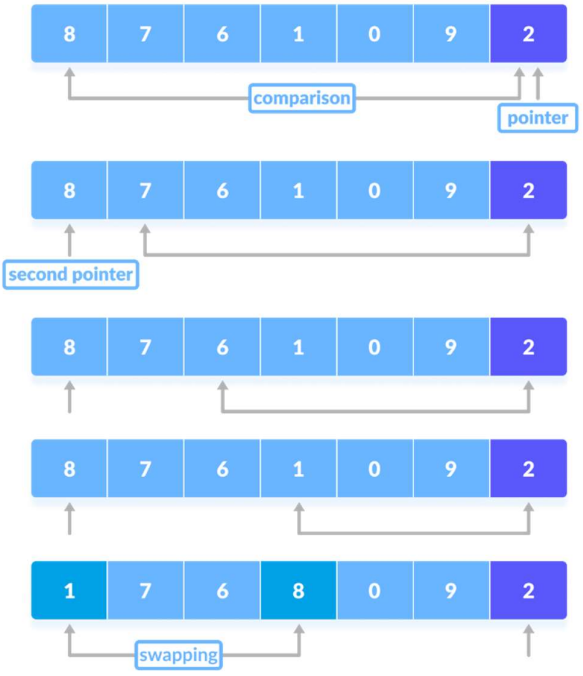
1. A pivot element is chosen from the array. You can choose any element from the array as the pivot element.
   Here, we have taken the rightmost of the array as the pivot element.

2. The elements smaller than the pivot element are put on the left and the elements greater than the pivot elements are put on the right.

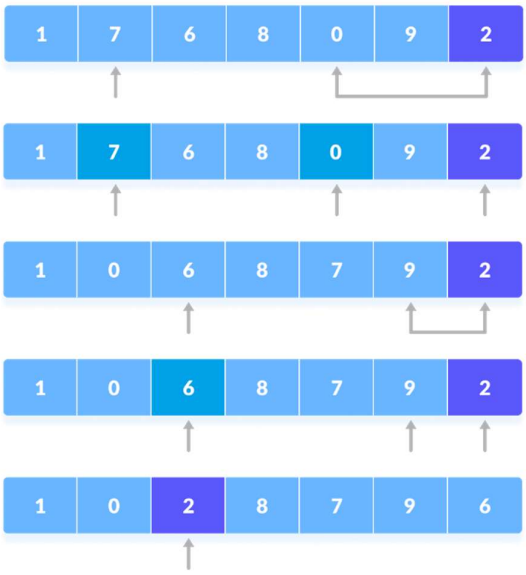
The above arrangement is achieved by the following steps,

- A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index. If the element greater than the pivot element is reached, a second pointer is set for that element.

- Now the pivot element is compared with the other elements (a third pointer). If an element smaller than the pivot element is reached, the smaller element is swapped with the grater element found earlier.

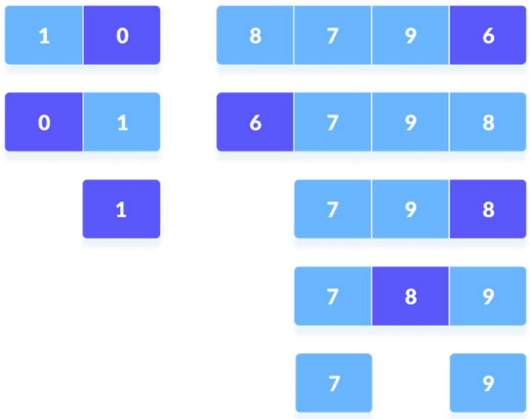


- The process goes on until the second last element is reached. Finally, the pivot element is swapped with the second pointer.



- Now the left and right subparts of this pivot element are taken for further processing in the steps below.

3. Pivot elements are again chosen for the left and the right subparts separately. Within these sub-parts, the pivot elements are placed at their right position. Then, step 2 is separated.



4. The sub-parts are again divided into smaller sub-parts until each subpart is formed of a single element.
5. At this point, the array is already sorted.

Applications

Quick sort is implemented when,

- The programming language is good for recursion
- Time complexity matters
- Space complexity matters

Counting Sort

Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Counting Sort Algorithm

```
countingSort (array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Complexity

- Worst case – O (n + k)
- Best case – O (n + k)
- Average case – O (n + k)
- Space complexity – O (max)

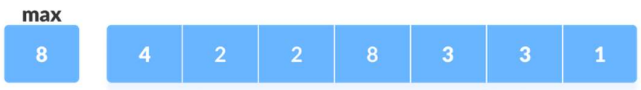
Applications

Counting sort is used when,

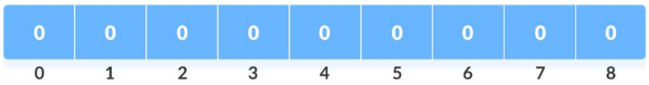
- There are smaller integers with multiple counts.
- Linear complexity is the need.

Example

1. Find out the maximum element (let it be **max**) from the given array.

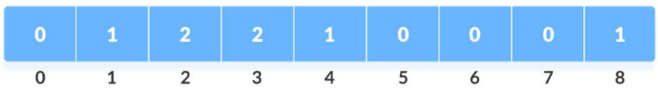


2. Initialize an array of length **(max+1)** with all elements 0. This array is used for storing the count of the elements in the array.

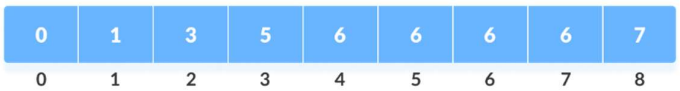


3. Store the count of each element at their respective index in **count** array.

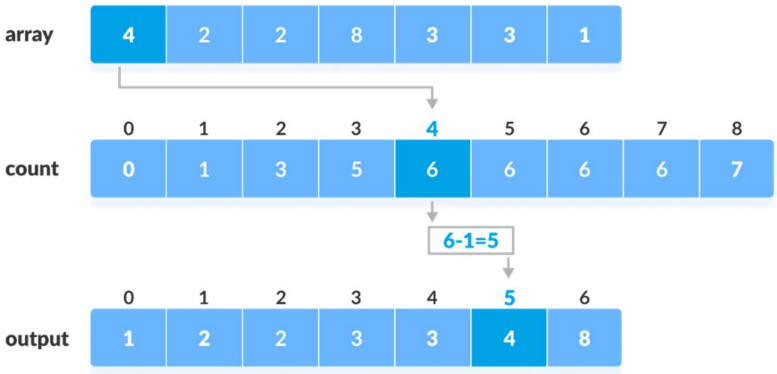
   For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of **count** array. If element "5" is not present in the array, then **0** is stored in 5th position.



4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as below.



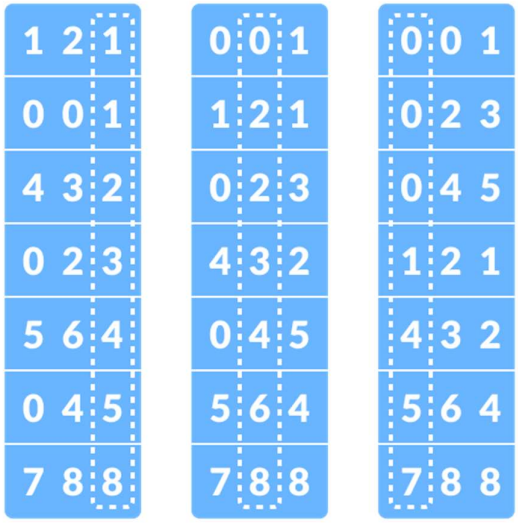
6. After placing each element at its correct position, decrease its count by one.

Radix Sort

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is stored according to radix sort as shown,



sorting the integers according to units, tens and hundreds place digits

Complexity of Radix Sort

The time complexity is **O (d (n + k))**

Here, d = number cycle

O (n + k) = time complexity of counting sort

Applications

Radix sort is implemented in,

- DC3 algorithm (Karkkainen-Sanders-Burkhardt) while making a suffix array.
- Places where there are numbers in large ranges.

Example

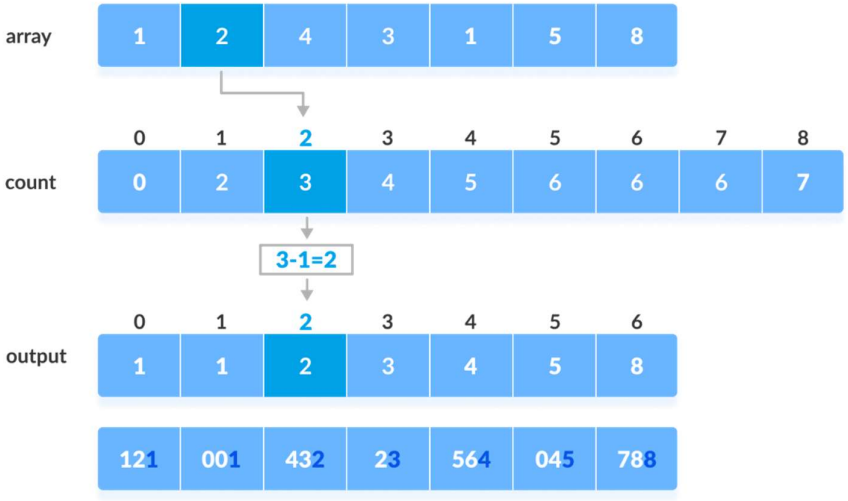
1. Find the largest element in the array, i.e., max. let x be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.

   In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).

2. Now, go through each significant place one by one.

   Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.

   Sort the elements based on the unit places digits. (x = 0).



3. Now, sort the elements based on digits at tens place.

4. Finally, sort the elements based on the digits at hundreds place.

Heap Sort

- Heap sort is a comparison-based sorting algorithm.
- Heap sort is similar to selection sort. The only difference is, it finds largest element and places the it at the end.
- This sort is not a stable sort. It requires a constant space for sorting a list.
- Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Complexity

Heap sort has **O (n long n)** time complexities for all the cases.

Applications

System concerned with security and embedded systems such as Linux Kernel use Heap Sort.

Example

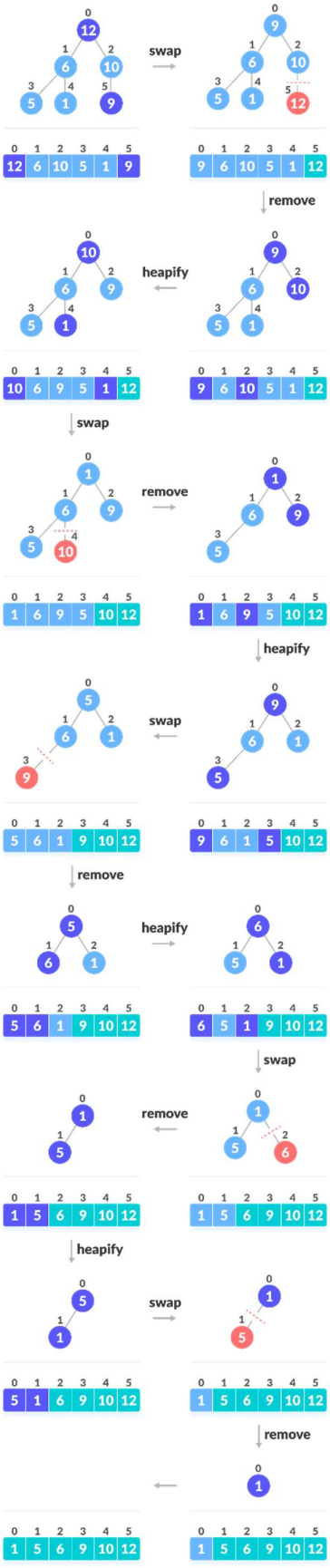
1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

2. **Swap**:
   Remove the root element and put at the end of the array (n^{th} position). Put the last item of the tree (heap) at the vacant place.

3. **Remove**:
   Reduce the size of the heap by 1.

4. **Heapify**:
   Heapify the root element again so that we have the highest element at root.

5. The process is repeated until all the items of the list are sorted.


swap
remove
heapify
swap
remove
heapify
swap
remove
heapify
swap
remove
heapify
swap
remove

Notes

1. The **Quick Sort** is the **best** sorting algorithm among all the sorting techniques. Because the time complexity of quick sort is O (n log n) in the best case, O (n log n) in average case and O (n^2) in the worst case. But because it has the best performance in the average case for most inputs,
   So, Quick Sort is generally considered as the fastest sorting algorithm.
2. The worst sorting algorithm is **Bogo Sort** among all.
3. **Linear search** is the best searching algorithm.

Data Structure & Algorithm Interview Questions

Difference between Heap & Stack memory?

The main difference between heap and stack is that **stack** is used to store local variables and function call while **heap** memory is used to store objects, no matter where the object is created in code.

Another difference between stack and heap is that size of stack memory is a lot lesser than the size of heap memory.

Variable stored in stacks are only visible to the owner thread while objects created in the heap are visible to all the thread. In other words, stack memory is kind of private memory of threads while heap memory is shared among all threads.

What is the difference between file structure & storage structure?

The main difference between file and storage structure is based on memory area that is being accessed.

- **Storage Structure:** It is the representation of the data structure in the computer memory.
- **File structure:** It is the representation of the storage structure in the auxiliary memory.

Which data structure is used to perform recursion?

Stack data structure is used in recursion due to its LIFO (Last in First Out) nature. Operating system maintains the stack in order to save the iteration variables at each function call.

What is the minimum no of queues that can be used to implement a priority Queue?

Two queues are needed. One queue is used to store the data elements, and another is used for storing priorities.

Which data structure suits most in the tree construction?

Queue Data Structure

What is a postfix expression?

An expression in which operators follow the operand is known as **postfix** is known as postfix.

The main benefit of this form is that there is no need to group sub-expression in parenthesis or to consider operator precedence.

The expression “a + b” will be represented as “ab+” in postfix notation.

Which notation is used in evaluation of arithmetic expression using prefix and postfix forms?

Polish and Reverse Polish Notation

How is an array different from linked list?

- The size of the array is fixed, linked lists are dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, whereas both insertion and deletion can easily be done in Linked list.
- Random access is not allowed in linked list.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have better cache locality that can make a pretty big difference in performance.

Which data structure are used for BFS and DFS of a graph?

- Queue is used for BFS
- Stack is used for DFS

What are the criteria for algorithm analysis?

An algorithm is generally analysed on two factors – **time** and **Space**, i.e., how much execution time and how much extra space required by the algorithm.

What are the common operations that can be performed on a data structure?

- Insertion – adding a data item
- Deletion – removing a data item
- Traversal – accessing or printing all data items
- Searching - finding a particular data item.
- Sorting – arranging data items in a pre-defined sequence.

Briefly explain the approaches to develop algorithms?

There are three commonly used approaches to develop algorithms,

1. **Greedy Approach:** Finding solution by choosing next best option.
2. **Divide & Conquer:** Dividing the problem to a minimum possible sub-problem and solving them independently.
3. **Dynamic Programming:** Dividing the problem to a minimum possible sub-problem and solving them combinedly.

How depth first traversal works?

Depth First Search algorithm (DFS) traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

How breadth search traversal works?

Breadth First Search algorithms (BFS) traverses a graph in a breadth wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

What is Tower of Hanoi?

Tower of Hanoi is a mathematical puzzle which consists of 3 tower (pegs) and more than one rings. All rings are of different size and stacked upon each other where the large disk is always below th small disk. The aim to move the tower of disk from one peg to another, without breaking its properties.

What are the major data structures used in the following areas: RDBMS, Network data model and Hierarchical data model?

- RDBMS – Arrays
- Network Data Model – Graph
- Hierarchical Data Model – Trees

Sorting is not possible by using which of the following methods?

(Insertion, Selection, Exchange, Deletion)

Sorting is not possible in **Deletion**.

Using **insertion,** we can perform insertion sort, using **selection** we can perform selection sort, using exchange we can perform the bubble sort (and other similar sorting methods). But no sorting method can be done just using deletion.

What are the methods available in sorting sequential files?

- Straight Merging
- Natural Merging
- Polyphase Sort
- Distribution of Initial Runs

List out few of the applications of tree data structure?

- The manipulation of Arithmetic Expression
- Symbol Table Construction
- Syntax Analysis

List out few of the applications that make uses of Multilinked Structures?

- Sparse Matrix
- Index Generation

What is the type of the algorithm used in solving the 8 Queens problem?

Backtracking

In an AVL tree, at which condition the balancing is to be done?

If the 'pivotal value' (Height Factor) is greater than 1 or less than -1.

What is the bucket size, when the overlapping and collision occurs at same time?

One.

If there is only one entry possible in the bucket, when the collision occurs, there is no way to accommodate the colliding value. This results in the overlapping of values.

Classify the hashing functions based on the various methods by which the key value is found?

1. Direct Method
2. Subtraction Method
3. Modulo-Division Method
4. Digit-Extraction Method
5. Mid-Square Method
6. Folding Method
7. Pseudo-Random Method

What are the types of collision resolution techniques and the methods used in each of the type?

- **Open addressing** (Closed hashing)
  The methods used include: Overflow Block
- **Closed addressing** (open hashing)
  The methods used include: Linked list, Binary tree

In RDBMS, what is the efficient data structure used in the internal storage representation?

B+ tree,

Because in B+ tree, all the data is stored in leaf nodes, that makes searching easier. This corresponds to the records that shall be stored in leaf nodes.

What is a Spanning tree?

A spanning tree is a tree associated with a network. All the nodes of the graph appear on the tree once. A minimum spanning tree is a spanning tree organized so that the total edge weight between nodes is minimized.

Does the minimum spanning tree of a graph give the shortest distance between any 2 specified nodes?

No,

The minimal spanning tree assures that the total weight of the tree is kept at its minimum. But it doesn't mean that the distance between any two nodes involved in the minimum spanning tree is minimum.

Which is the simplest file structure?

Sequential is the simplest file structure.

Whether Linked List is Linear or Non-linear data structure?

According to access strategies Linked List is a Linear one.

According to Storage Linked List is a Non-linear one.

Is it possible to find a loop in a Linked list?
a. Possible at O(n)
b. Not possible
c. Possible at O(n^2) only
d. Depends on the position of loop

Solution: a. Possible at O(n)
Have two pointers say P1 and P2 pointing to the first node of the list.
Start a loop and Increment P1 once and P2 twice in each iteration. At any point of time if P1==P2 then there is a loop in that linked list. If P2 reaches NULL (end of linked list) then no loop exists.

Two linked lists L1 and L2 intersects at a particular node N1 and from there all other nodes till the end are common. The length of the lists are not same. What are the possibilities to find N1?.

a. Solution exist for certain cases only
b. No linear solution exists
c. Linear solution is possible
d Only Non-linear solution exist.

Solution: c. Linear solution is possible
Have two pointers say P1 pointing to the first node of L1 and P2 to that of L2. Traverse through both the lists. If P1 reaches L1's last node, point it to the first node of L2 and continue traversing. Do the same thing for P2 when it reaches L2's last node. (By doing this, we are balancing the difference in the length between the linked lists. The shorter one will get over soon and by redirecting to longer list's head, it will traverse the extra nodes also.) Finally, they will Meet at the Intersection node.

The below method 'PrintTree' results in which of the following traversal
a Inorder
b. Preorder
c. Postorder
d. None of the above

```
void PrintTree (Tree T)
{
if (T != NULL)
{
PrintTree (T-> Left);
PrintElement (T-> Element);
PrintTree (T->Right);
}
}
```

Solution: a. Inorder
Inorder:

```
void PrintTree (Tree T)
{
if (T != NULL)
{
PrintTree (T-> Left);
PrintElement (T-> Element);
PrintTree (T->Right);
}
}
```

For preorder use this order

```
PrintElement (T-> Element);
PrintTree (T-> Left);
PrintTree (T->Right);
```

For postorder use this order

```
PrintTree (T-> Left);
PrintTree (T->Right);
PrintElement (T-> Element);
```

Given a Binary Search Tree (BST), print its values in ascending order.

a. Perform Depth first traversal
b. Perform Breadth first traversal
c. Perform Postorder traversal
d. Perform Inorder traversal

Solution: d. Perform Inorder traversal
It is the property of BST and Inorder traversal.

Is it possible to implement a queue using Linked List? Enqueue & Dequeue should be O (1).

a. Not possible to implement.
b Only Enqueue is possible at O(1).
c. Only Dequeue is possible at O(1).
d. Both Enqueue and Dequeue is possible at O(1)

Solution: d. Both Enqueue and Dequeue is possible at O(1)
Have two pointers H pointing to the Head and T pointing to the Tail of the linked list. Perform enqueue at T and perform dequeue at H. Update the pointers after each operation accordingly.

Given a Tree, is it possible to find the greatest and least among leaves in linear time?

a. Solution depends on the tree structure
b. Linear solution exist
c. Only Non-linear solution exist.
d. No linear solution exists

Solution: b. Linear solution exist
Have two variables Min and Max. Perform any tree traversal. Assign the first traversed leaf element to Min and Max for all other leaf elements check with these variables and update it accordingly. If a current element is < Min then updates Min with that element. If it is > Min then checks with Max.
Note: If you want to find the greatest and least among all nodes perform the checks for each node traversed.

Is it possible to find the greatest and least value among the nodes in a given BST without using any extra variables?

a. No solution exists.
b. Solution need 2 extra variables
c. Solution exist without any extra variables
d Solution need 1 extra variable

Solution: c Solution exist without any extra variables
As per BST property, the left most node should be the least one and the rightmost node should be the greatest. In other words, the first and last node of an Inorder traversal are the least and greatest among the nodes respectively.

Is it possible to implement 2 stacks in an array?

Condition: None of the stack should indicate an overflow until every slot of an array is used.
a. Only 1 stack can be implemented for the given condition
b. Stacks cannot be implemented in array
c. 2 stacks can be implemented for the given condition.
d. 2 stacks can be implemented if the given condition is applied only for 1 stack.

Solution: c. 2 stacks can be implemented for the given condition
Start 1st stack from left (1st position of an array) and 2nd from right (last position say n). Move 1st stack towards right (i.e., 1,2,3 ...n) and 2nd towards left (i.e., n, n-1, n-2...1).

Given two keys K1 & K2, write an algorithm to print all the elements between them with K1<=K2 in a BST.

a. Solution need 2 extra spaces
b. Linear solution is possible without using any extra space
c No linear solution exists
d Solution need 1 extra space

Solution: b. Linear solution is possible without using any extra space
Perform an inorder traversal. Once you find K1 print it and continue traversal now, print all other traversed elements until you reach K2.
Note: If K1 == K2 stop once you find K1.

How many stacks are required to implement a Queue?

a. One
b. Two
c. Three
d. Two + one extra space.

Solution: b Two
Have two stacks S1 and S2.
For Enqueue, perform push on S1.
For Dequeue, if S2 is empty pop all the elements from S1 and push it to S2. The last element you popped from S1 is an element to be dequeued. If S2 is not empty, then pop the top element in it.