Data Structures through C

Contents

- Introduction to Data structures
- Searching
- Sorting
- Linked Lists
- Linear Data Structures
  - Stacks
  - Queues
- Non-Linear Data Structures-Trees

Introduction

- A data structure is a storage that is used to store and organize data.
- It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
- A data structure is not only used for organizing the data.
- It is also used for processing, retrieving, and storing data.
- There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed.

Data structures provide an easy way of organizing, retrieving, managing, and storing data. Here is a list of the needs for data.

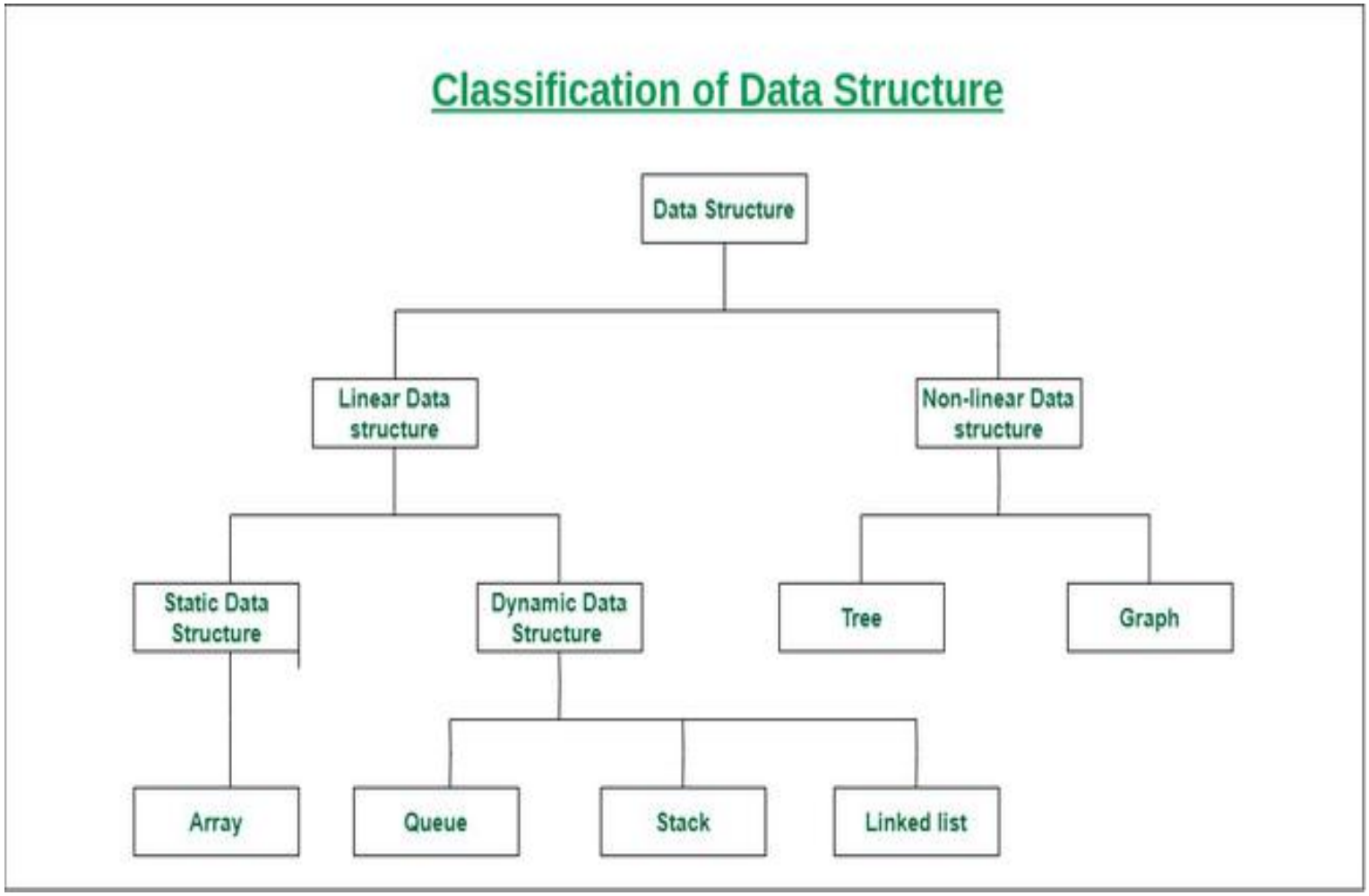
Efficient data access and manipulation: Data structures enable quick access and manipulation of data. For example, an array allows constant-time access to elements using their index, while a hash table allows fast access to elements based on their key. Without data structures, programs would have to search through data sequentially, leading to slow performance.

Memory management: Data structures allow efficient use of memory by allocating and deallocating memory dynamically. For example, a linked list can dynamically allocate memory for each element as needed, rather than allocating a fixed amount of memory upfront. This helps avoid memory wastage and enables efficient memory management.

Code reusability: Data structures can be reused across different programs and projects. For example, a generic stack data structure can be used in multiple programs that require LIFO (Last-In-First-Out) functionality, without having to rewrite the same code each time.

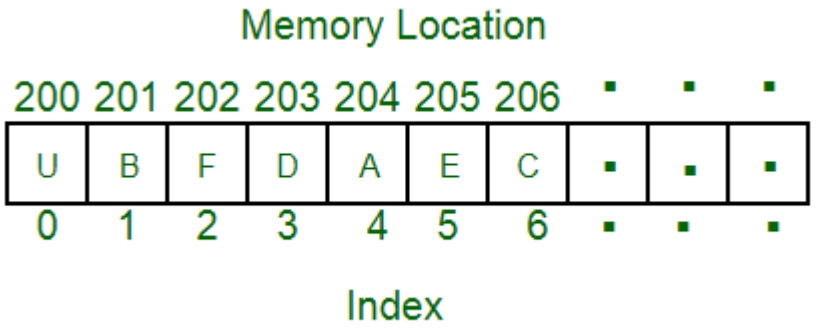
Optimization of algorithms: Data structures help optimize algorithms by enabling efficient data access and manipulation. For example, a binary search tree allows fast searching and insertion of elements, making it ideal for implementing searching and sorting algorithms.

Scalability: Data structures enable programs to handle large amounts of data effectively. For example, a hash table can store large amounts of data while providing fast access to elements based on their key.


Classification of Data Structure
Data Structure
Linear Data structure
Non-linear Data structure
Static Data Structure
Dynamic Data Structure
Tree
Graph
Array
Queue
Stack
Linked list

Array:

- An array is a collection of data items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
- This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Applications of Array Data Structure:

Below are some applications of arrays.

Storing and accessing data: Arrays are used to store and retrieve data in a specific order. For example, an array can be used to store the scores of a group of students, or the temperatures recorded by a weather station.
Sorting: Arrays can be used to sort data in ascending or descending order. Sorting algorithms such as bubble sort, merge sort, and quicksort rely heavily on arrays.
Searching: Arrays can be searched for specific elements using algorithms such as linear search and binary search.
Matrices: Arrays are used to represent matrices in mathematical computations such as matrix multiplication, linear algebra, and image processing.
Stacks and queues: Arrays are used as the underlying data structure for implementing stacks and queues, which are commonly used in algorithms and data structures.
Graphs: Arrays can be used to represent graphs in computer science. Each element in the array represents a node in the graph, and the relationships between the nodes are represented by the values stored in the array.
Dynamic programming: Dynamic programming algorithms often use arrays to store intermediate results of subproblems in order to solve a larger problem.

Below are some real-time applications of arrays.

Signal Processing: Arrays are used in signal processing to represent a set of samples that are collected over time. This can be used in applications such as speech recognition, image processing, and radar systems.

Multimedia Applications: Arrays are used in multimedia applications such as video and audio processing, where they are used to store the pixel or audio samples. For example, an array can be used to store the RGB values of an image.

Data Mining: Arrays are used in data mining applications to represent large datasets. This allows for efficient data access and processing, which is important in real-time applications.

Robotics: Arrays are used in robotics to represent the position and orientation of objects in 3D space. This can be used in applications such as motion planning and object recognition.

Real-time Monitoring and Control Systems: Arrays are used in real-time monitoring and control systems to store sensor data and control signals. This allows for real-time processing and decision-making, which is important in applications such as industrial automation and aerospace systems.

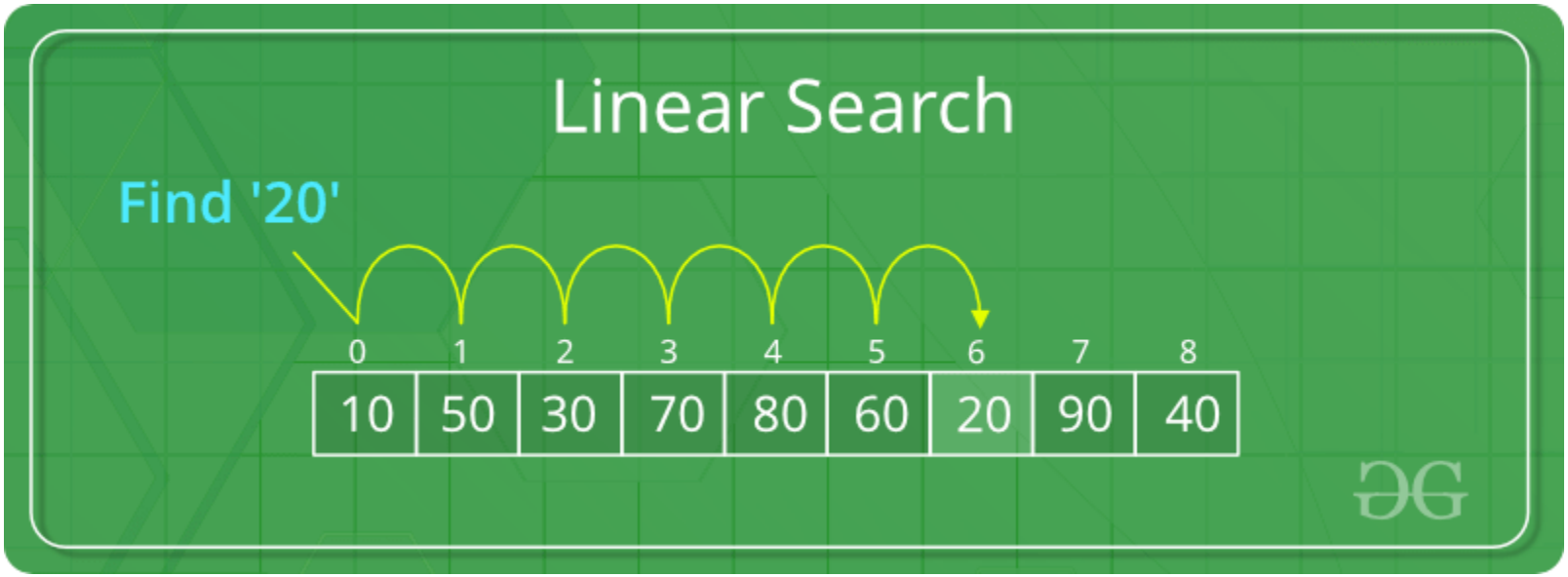
Financial Analysis: Arrays are used in financial analysis to store historical stock prices and other financial data. This allows for efficient data access and analysis, which is important in real-time trading systems.

Scientific Computing: Arrays are used in scientific computing to represent numerical data, such as measurements from experiments and simulations. This allows for efficient data processing and visualization, which is important in real-time scientific analysis and experimentation.

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.

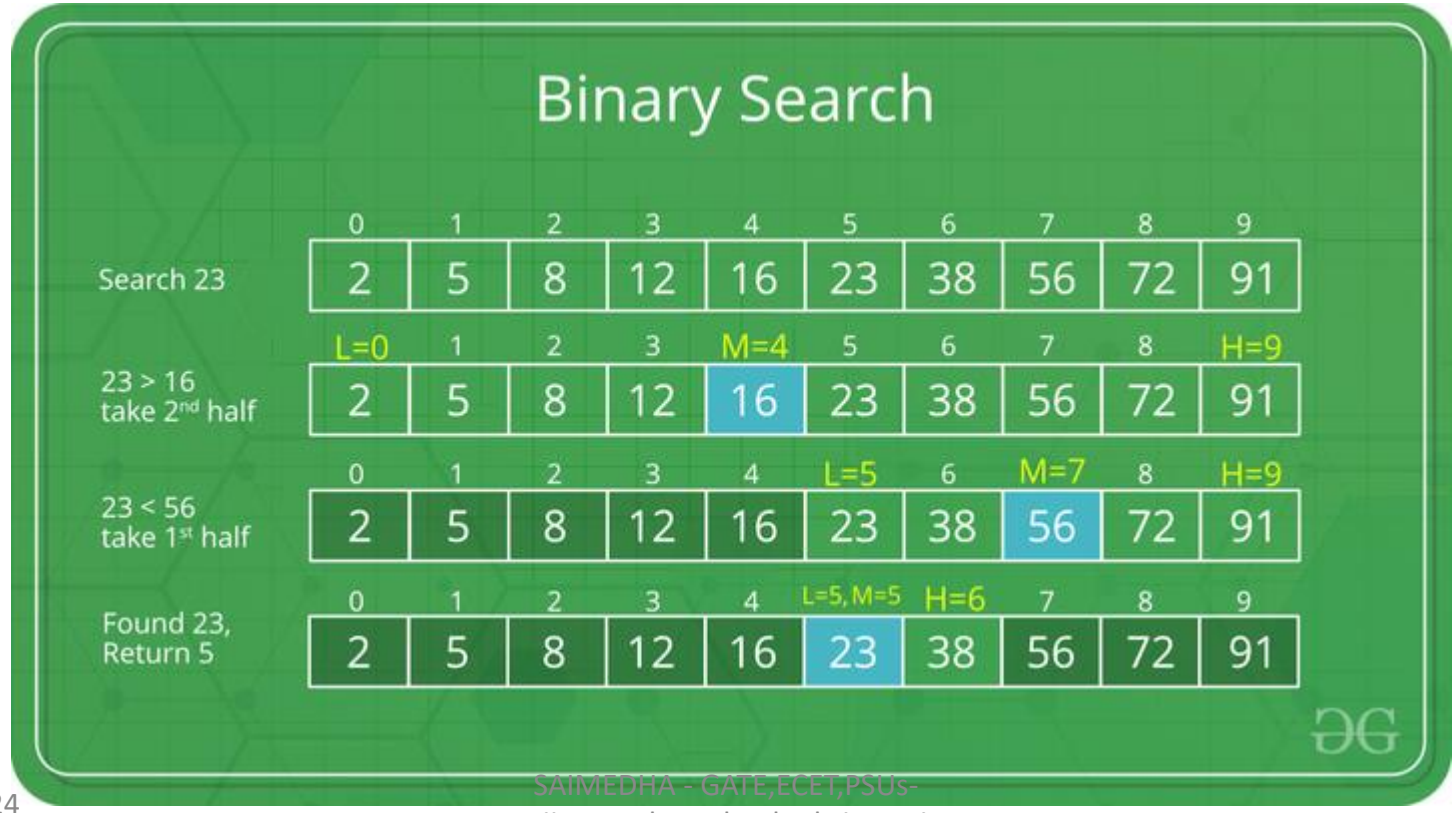
Linear Search to find the element “20” in a given list of numbers



Interval Search:

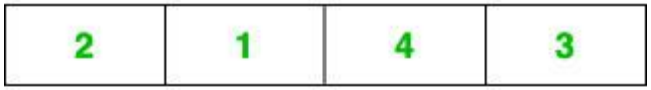
- These algorithms are specifically designed for searching in sorted data-structures.
- These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half.
- For Example: Binary Search.

Binary Search to find the element “23” in a given list of numbers

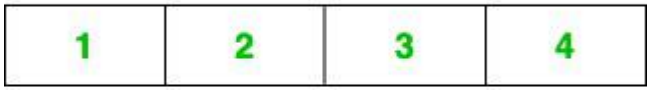


Basis of comparison	Linear search	Binary search
Definition	The linear search starts searching from the first element and compares each element with a searched element till the element is not found.	It finds the position of the searched element by finding the middle element of the array.
Sorted data	In a linear search, the elements don't need to be arranged in sorted order.	The pre-condition for the binary search is that the elements must be arranged in a sorted order.
Implementation	The linear search can be implemented on any linear data structure such as an array, linked list, etc.	The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.
Approach	It is based on the sequential approach.	It is based on the divide and conquer approach.
Size	It is preferrable for the small-sized data sets.	It is preferrable for the large-size data sets.
Efficiency	It is less efficient in the case of large-size data sets.	It is more efficient in the case of large-size data sets.
Worst-case scenario	In a linear search, the worst- case scenario for finding the element is O(n).	In a binary search, the worst-case scenario for finding the element is $O(\log_2 n)$.
Best-case scenario	In a linear search, the best-case scenario for finding the first element in the list is O(1).	In a binary search, the best-case scenario for finding the first element in the list is O(1).
Dimensional array	It can be implemented on both a single and multidimensional array.	It can be implemented only on a multidimensional array.

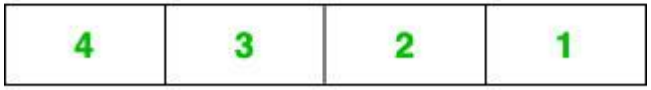
A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.



Unsorted Array



Array sorted in ascending order



Array sorted in descending order

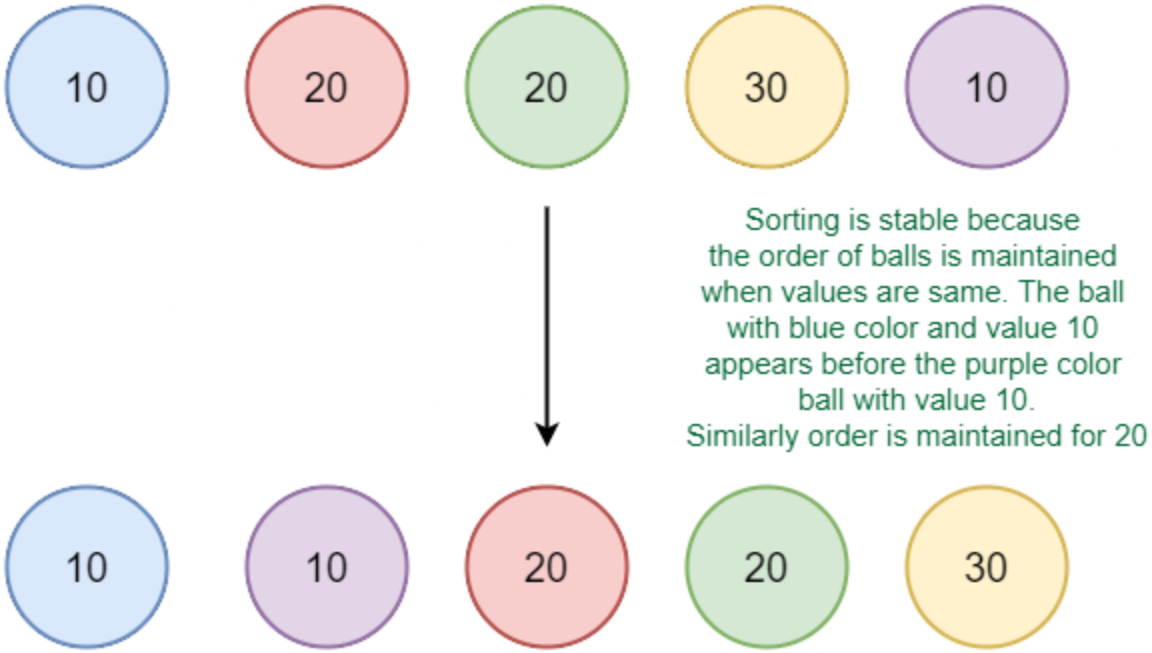
Sorting Algorithms:

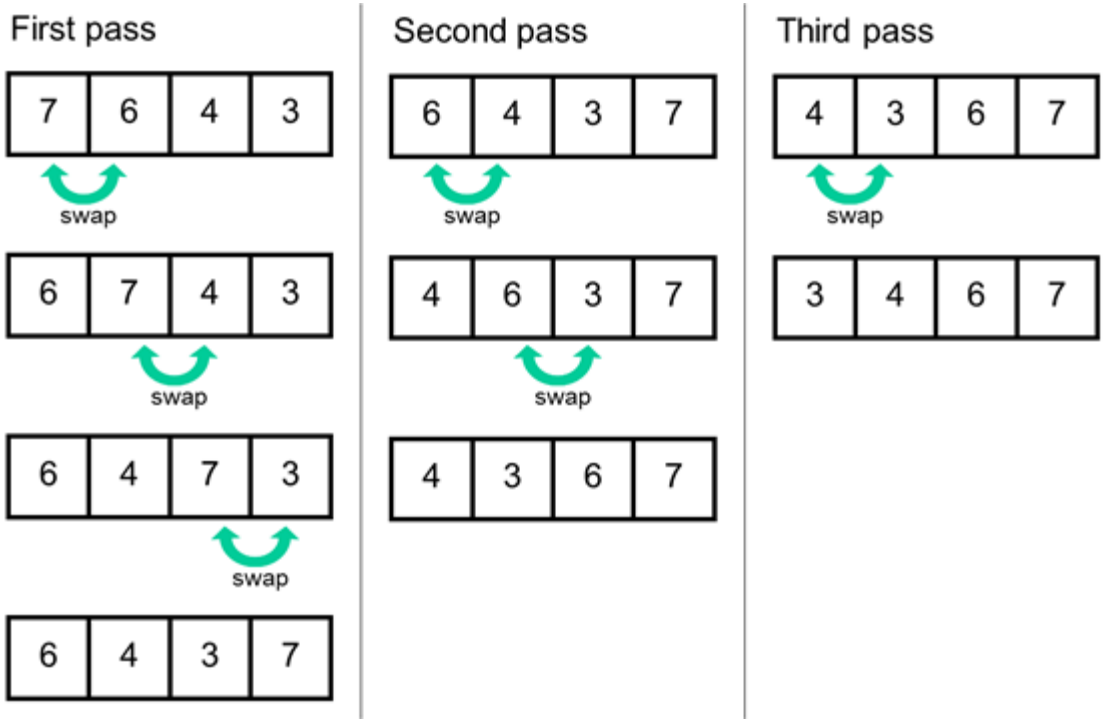
- Bubble sort (sorting by exchange, comparison based sort)
- Insertion sort
- Selection sort
- Merge sort
- Quick sort

Bubble Sort

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
- Bubble sort is stable sort and in-place sort.
- In-place means that the algorithm does not use extra space for manipulating the input but may require a small non-constant extra space for its operation. Usually, this space is O(log n), though sometimes anything in O(n) (Smaller than linear) is allowed.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set.





If we have total n elements, then we need to repeat this process for n-1 times (n-1 passes).

It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

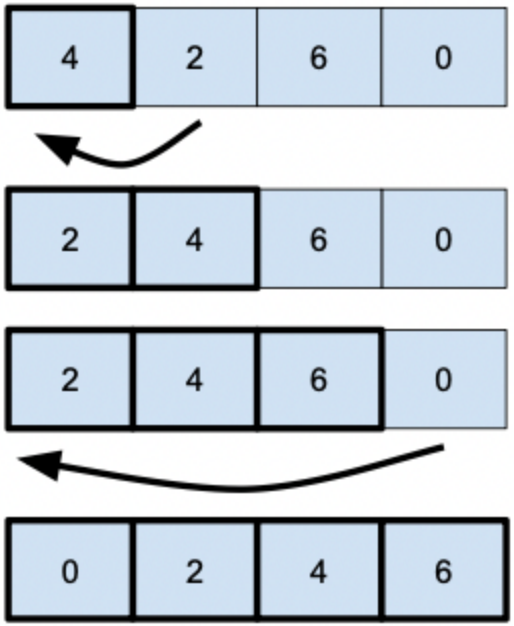
Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is O(n^2).

Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is O(n^2).

Insertion sort

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.
- insertion sort is an in-place sorting algorithm
- insertion sort is a stable sorting algorithm



An insertion algorithm consists of N-1 passes when an array of N elements is given.

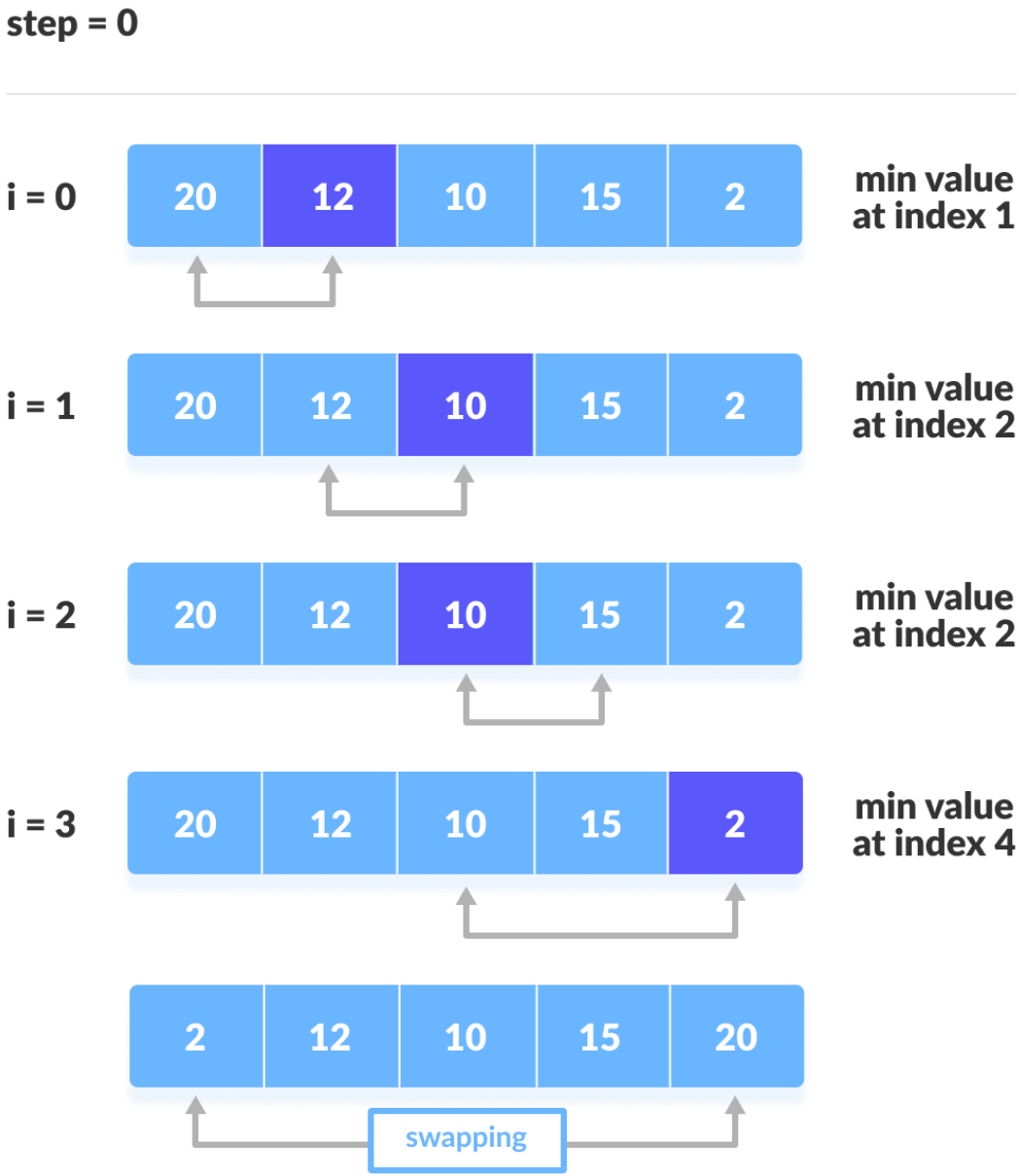
Best Case Complexity: The insertion sort algorithm has a best-case time complexity of O(n) for the already sorted array.

Average Case Complexity: The average-case time complexity for the insertion sort algorithm is O(n^2), which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also O(n^2), which occurs when we sort the ascending order of an array into the descending order.

Selection Sort

- Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion.
- This process is repeated for the remaining unsorted portion of the list until the entire list is sorted.


step = 0
i = 0
20
12
10
15
2
min value at index 1
i = 1
20
12
10
15
2
min value at index 2
i = 2
20
12
10
15
2
min value at index 2
i = 3
20
12
10
15
2
min value at index 4
2
12
10
15
20
swapping

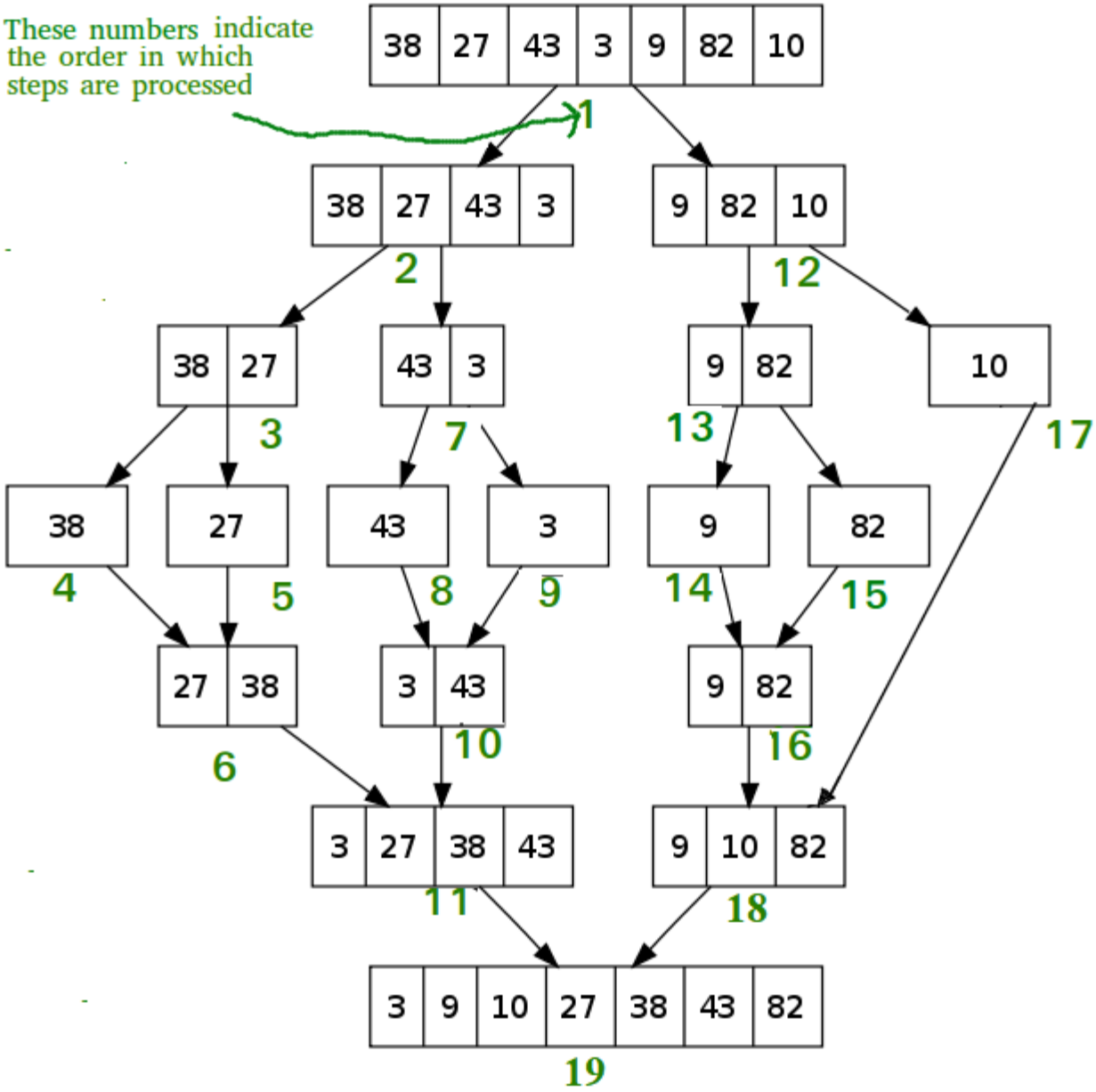
Best Case Complexity: The selection sort algorithm has a best-case time complexity of O(n2) for the already sorted array.

Average Case Complexity: The average-case time complexity for the selection sort algorithm is O(n2), in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also O(n2), which occurs when we sort the descending order of an array into the ascending order.

Merge sort

- **Merge sort** is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
4
5
8
9
14
15
27 38
3 43
9 82
10
16
6
3 27 38 43
9 10 82
11
18
3 9 10 27 38 43 82
19

Is Merge sort In Place?

No, In merge sort the merging step requires extra space to store the elements.

Is Merge sort Stable?

Yes, merge sort is stable.

Best Case Complexity: The merge sort algorithm has a best-case time complexity of O(n*log n) for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is O(n*log n), which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also O(n*log n), which occurs when we sort the descending order of an array into the ascending order.

Quick sort

- Quick sort, also known as partition-exchange sort, is an in-place sorting algorithm.
- It is a divide-and-conquer algorithm that works on the idea of selecting a pivot element and dividing the array into two subarrays around that pivot.
- In quick sort, after selecting the pivot element, the array is split into two subarrays.
- One subarray contains the elements smaller than the pivot element, and the other subarray contains the elements greater than than the pivot element.

Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n*logn).

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is O(n*logn).

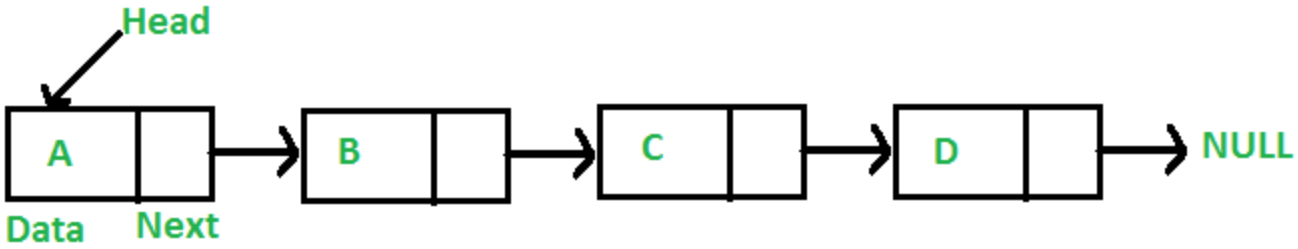
Worst Case Complexity - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is O(n^2).

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)

Sorting Algorithms	In - Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Quick Sort	Yes	No
Merge Sort	No (because it requires an extra array to merge the sorted subarrays)	Yes
Heap Sort	Yes	No

Linked Lists:

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Why linked list data structure needed?

Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.

Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.

Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Applications of Linked Lists:

- Linked Lists are used to implement stacks and queues.
- It is used for the various representations of trees and graphs.
- It is used in dynamic memory allocation(linked list of free blocks).
- It is used for representing sparse matrices.
- It is used for the manipulation of polynomials.
- It is also used for performing arithmetic operations on long integers.
- It is used for finding paths in networks.
- In operating systems, they can be used in Memory management, process scheduling and file system.
- Linked lists can be used to improve the performance of algorithms that need to frequently insert or delete items from large collections of data.
- Implementing algorithms such as the LRU cache, which uses a linked list to keep track of the most recently used items in a cache.

Applications of Linked Lists in real world:

- The list of songs in the music player are linked to the previous and next songs.
- In a web browser, previous and next web page URLs are linked through the previous and next buttons.
- In image viewer, the previous and next images are linked with the help of the previous and next buttons.
- Switching between two applications is carried out by using “alt+tab” in windows and “cmd+tab” in mac book. It requires the functionality of circular linked list.
- In mobile phones, we save the contacts of the people. The newly entered contact details will be placed at the correct alphabetical order. This can be achieved by linked list to set contact at correct alphabetical position.
- The modifications that we make in documents are actually created as nodes in doubly linked list. We can simply use the undo option by pressing Ctrl+Z to modify the contents. It is done by the functionality of linked list.

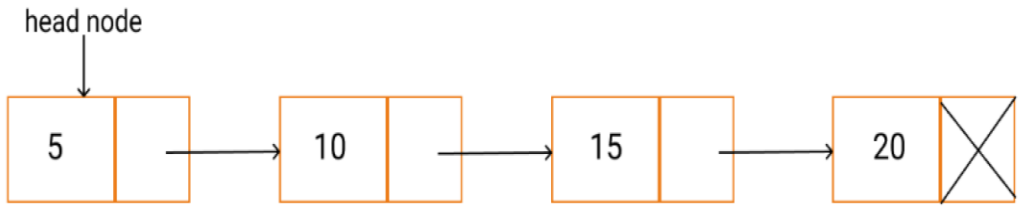
Advantages of Linked Lists:

- Linked lists are a popular data structure in computer science, and have a number of advantages over other data structures, such as arrays. Some of the key advantages of linked lists are:

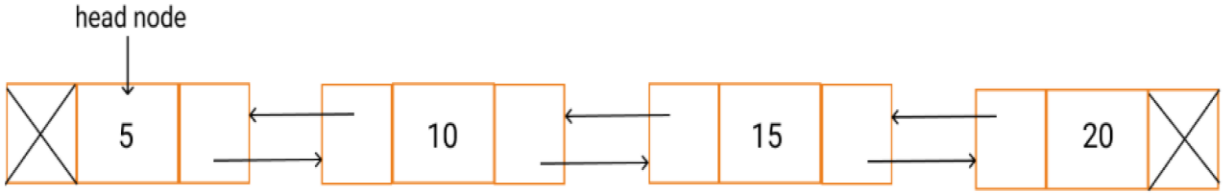
- Dynamic size: Linked lists do not have a fixed size, so you can add or remove elements as needed, without having to worry about the size of the list. This makes linked lists a great choice when you need to work with a collection of items whose size can change dynamically.
- Efficient Insertion and Deletion: Inserting or deleting elements in a linked list is fast and efficient, as you only need to modify the reference of the next node, which is an O(1) operation.
- Memory Efficiency: Linked lists use only as much memory as they need, so they are more efficient with memory compared to arrays, which have a fixed size and can waste memory if not all elements are used.
- Easy to Implement: Linked lists are relatively simple to implement and understand compared to other data structures like trees and graphs.
- Flexibility: Linked lists can be used to implement various abstract data types, such as stacks, queues, and associative arrays.
- Easy to navigate: Linked lists can be easily traversed, making it easier to find specific elements or perform operations on the list.

Types of Linked List

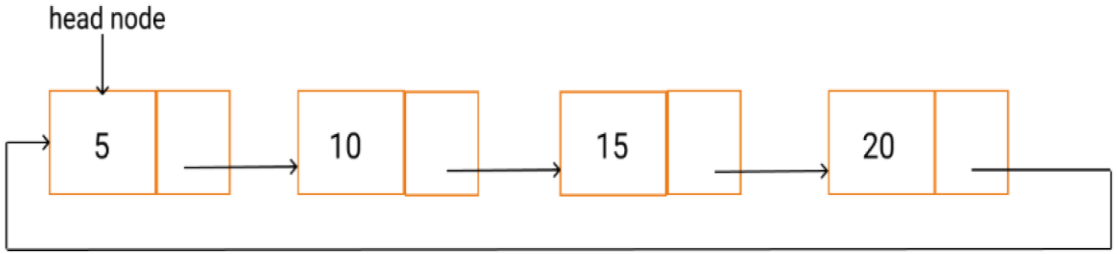
Singly Linked List



Doubly Linked List



Circular Linked List

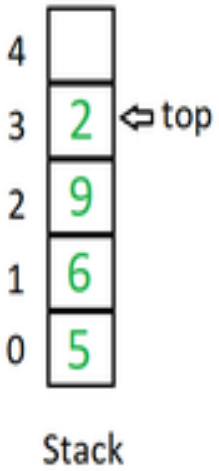


Disadvantages of Linked Lists:

- Linked lists are a popular data structure in computer science, but like any other data structure, they have certain disadvantages as well. Some of the key disadvantages of linked lists are:
- Slow Access Time: Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for, which is an O(n) operation. This makes linked lists a poor choice for situations where you need to access elements quickly.
- Pointers: Linked lists use pointers to reference the next node, which can make them more complex to understand and use compared to arrays. This complexity can make linked lists more difficult to debug and maintain.
- Higher overhead: Linked lists have a higher overhead compared to arrays, as each node in a linked list requires extra memory to store the reference to the next node.
- Cache Inefficiency: Linked lists are cache-inefficient because the memory is not contiguous. This means that when you traverse a linked list, you are not likely to get the data you need in the cache, leading to cache misses and slow performance.
- Extra memory required: Linked lists require an extra pointer for each node, which takes up extra memory. This can be a problem when you are working with large data sets, as the extra memory required for the pointers can quickly add up.

Stack:

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).
- In stack, all insertion and deletion are permitted at only one end of the list.



Stack

Mainly the following operations are performed in the stack:

Initialize: Make a stack empty.

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of the stack.

isEmpty: Returns true if the stack is empty, else false.

Application of Stack Data Structure:

Function calls and recursion: When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.

Undo/Redo operations: The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.

Expression evaluation: Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.

Browser history: Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.

Balanced Parentheses: Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.

Backtracking Algorithms: The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack

Application of Stack in real life:

- CD/DVD stand.
- Stack of books in a book shop.
- Call center systems.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).
- Allocation of memory by an operating system while executing a process.

Advantages of Stack:

Easy implementation: Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
Efficient memory utilization: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
Fast access time: Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
Helps in function calls: Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.
Supports backtracking: Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
Used in Compiler Design: Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
Enables undo/redo operations: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

Push:

Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
 increment top
 stack[top] assign value
end else
end procedure
```

Pop:

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Algorithm for pop:

```
begin
 if stack is empty
    return
 endif
else
 store value of stack[top]
 decrement top
 return value
end else
end procedure
```

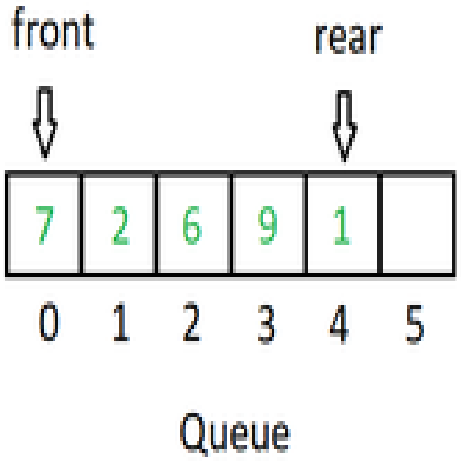
Infix to Postfix conversion algorithm:

- Scan all the symbols one by one from left to right in the given Infix Expression.
- If the reading symbol is operand, then immediately append it to the Postfix Expression .
- If the reading symbol is left parenthesis '(', then Push it onto the Stack.
- If the reading symbol is right parenthesis ')', then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
- If the reading symbol is operator (+ , – , * , /), then Push it onto the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix . If open parenthesis is there on top of the stack then push the operator into the stack.
- If the input is over, pop all the remaining symbols from the stack and append them to the postfix .

Postfix expression evaluation:

- Create a stack that holds integer type data to store the operands of the given postfix expression. Let it be **st.**
- Iterate over the string from left to right and do the following -
  - If the current element is an operand, push it into the stack.
  - Otherwise, if the current element is an operator (say /)do the following -
    - Pop an element from st, let it be op1.
    - Pop another element from st, let it be op2.
    - Computer the result of op2 / op1, and push it into the stack. Note the order .*i.e*. op2 / op1 should not be changed otherwise it will affect the final result in some cases.
- At last, st will consist of a single element .*i.e*. the result after evaluating the postfix expression.

Queue: Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). In the queue, items are inserted at one end and deleted from the other end. A good example of the 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 a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



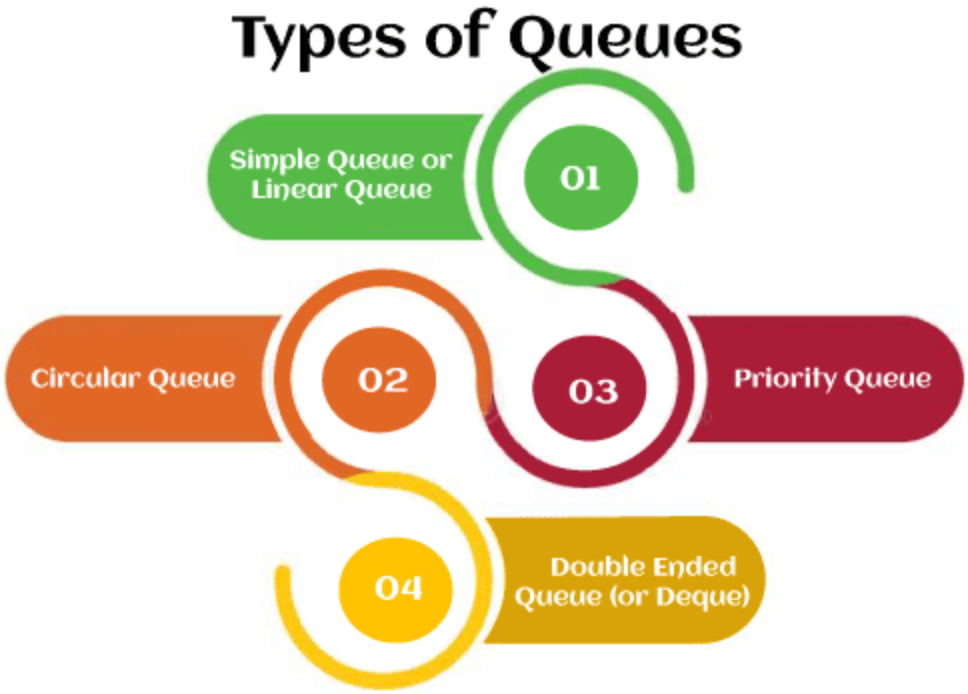
Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from the queue.

Rear: Get the last item from the queue


Types of Queues
Simple Queue or Linear Queue
01
Circular Queue
02
03
Priority Queue
04
Double Ended Queue (or Deque)

Simple Queue or Linear Queue

- In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.
- The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.
- It strictly follows the FIFO rule.

Drawback:

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Enqueue Algorithm

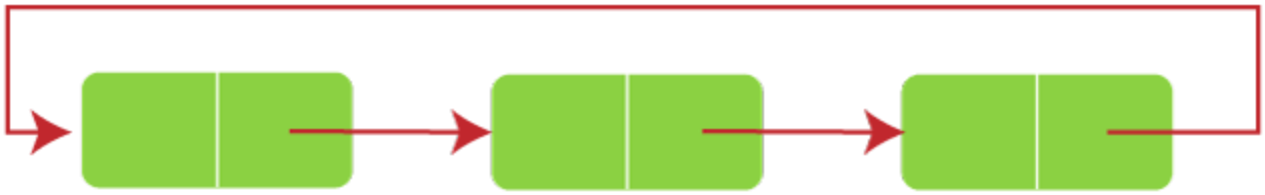
Step 1: IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT

Dequeue Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
Step 2: EXIT

Circular Queue

- In Circular Queue, all the nodes are represented as circular.
- It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- It is also known as Ring Buffer, as all the ends are connected to another end.

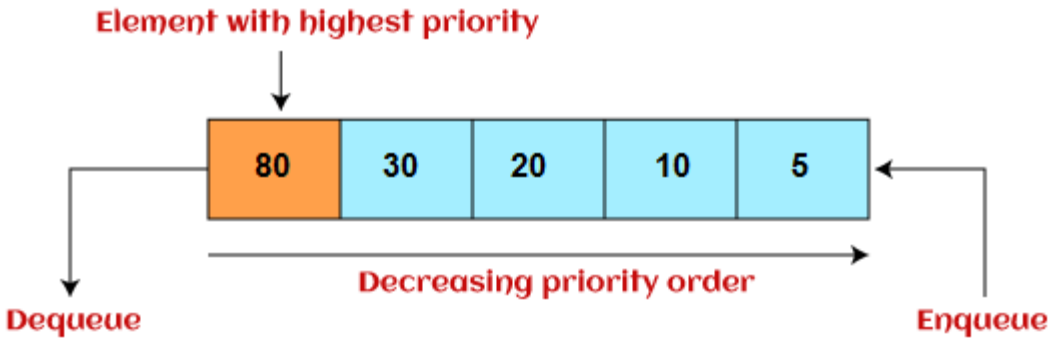


Circular Queue

The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Priority Queue

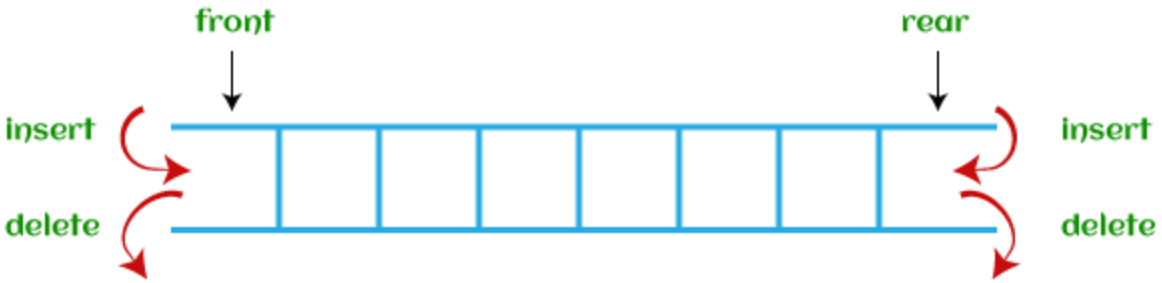
It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle.



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

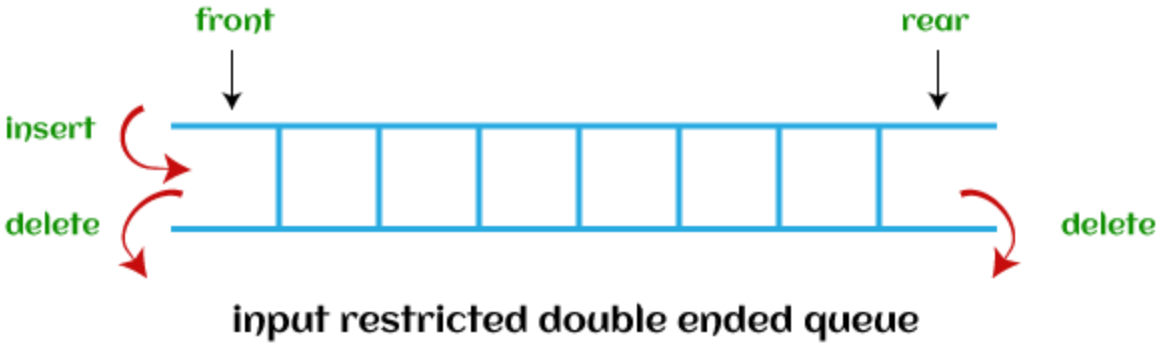
Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear.
It means that we can insert and delete elements from both front and rear ends of the queue.
Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends.
Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

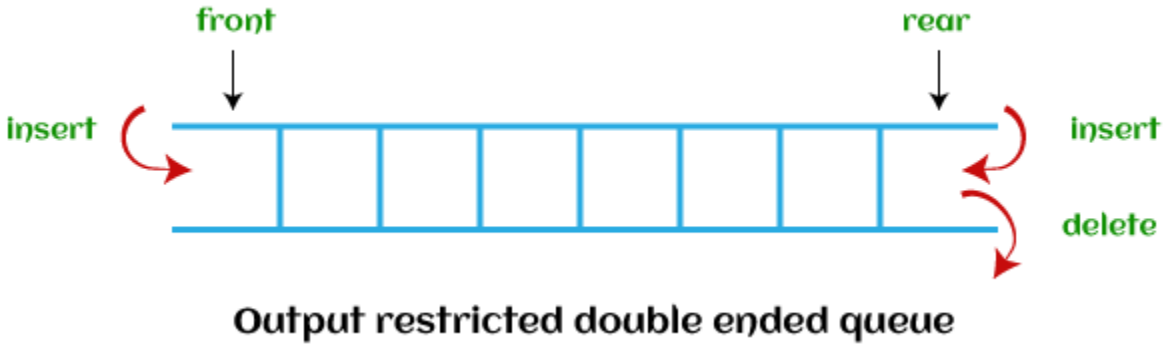


double ended queue

Input restricted deque - As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Output restricted deque - As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Applications of Queue in Operating systems:

- Semaphores
- FCFS (first come first serve) scheduling, example: FIFO queue
- Spooling in printers
- Buffer for devices like keyboard
- CPU Scheduling
- Memory management

Applications of Queue in Networks:

- Queues in routers/ switches
- Mail Queues
- Variations: (Deque, Priority Queue, Doubly Ended Priority Queue)

Some other applications of Queue:

- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp.
- Traffic software (Each light gets on one by one after every time of interval of time.)