GATE Data Science and AI Study Materials

Data Structures using Python

by Piyush Wairale

Instructions:

- Kindly go through the lectures/videos on our website www.piyushwairale.com
- Read this study material carefully and make your own handwritten short notes. (Short notes must not be more than 5-6 pages)
- Attempt the mock tests available on portal.
- Revise this material at least 5 times and once you have prepared your short notes, then revise your short notes twice a week
- If you are not able to understand any topic or required a detailed explanation and if there are any typos or mistake in study materials. Mail me at piyushwairale100@gmail.com

Contents

1 Built-In Data Structures **4**
- 1.1 List . . . 4
- 1.2 Tuples . . . 6
- 1.3 Sets . . . 7
- 1.4 Dictionaries . . . 8
- 1.5 Comparison of List, Tuple, Set, and Dictionary in Python . . . 8

2 Linked Lists **10**
- 2.1 Types of Linked Lists: . . . 11
  - 2.1.1 Singly Linked List: . . . 11
  - 2.1.2 Doubly Linked List: . . . 11
  - 2.1.3 Circular Single Linked Lists . . . 11
  - 2.1.4 Circular Double Linked Lists . . . 12
- 2.2 Python Implementation of a Singly Linked List . . . 12
- 2.3 Complexity Analysis . . . 14

3 Stacks and Queues **16**
- 3.1 Stacks . . . 16
- 3.2 Queues . . . 19
  - 3.2.1 Circular Queue . . . 21
  - 3.2.2 Priority Queue . . . 21
- 3.3 Implementing Stacks and Queues Using Linked Lists . . . 21

4 Trees **23**
- 4.1 Binary Trees . . . 24
- 4.2 Binary Search Trees (BST) . . . 25
- 4.3 Tree Traversal Methods . . . 26
  - 4.3.1 Example Usage . . . 26
- 4.4 Balancing Binary Search Trees . . . 27

5 Heaps (Optional for GATE DA) **28**
- 5.1 Heap Sort Algorithm . . . 29

6 Hashing **29**
- 6.1 Time Complexity of Hash Table Operations . . . 30
- 6.2 Load Factor . . . 30

7 Searching Algorithms **31**
- 7.1 Linear Search . . . 31
- 7.2 Binary Search . . . 32
- 7.3 Comparison: Linear Search vs. Binary Search . . . 33

8 Basic Sorting Algorithms **34**
- 8.1 Selection Sort . . . 34
- 8.2 Bubble Sort . . . 35
- 8.3 Insertion Sort . . . 35

9 Divide and Conquer Algorithms: Mergesort and Quicksort **37**
- 9.1 Mergesort . . . 37
- 9.2 Quicksort . . . 38
- 9.3 Comparison of Sorting Algorithms . . . 39

10 Previous Year Questions of GATE DA 2024 **40**

LinkedIn

Youtube Channel

Instagram

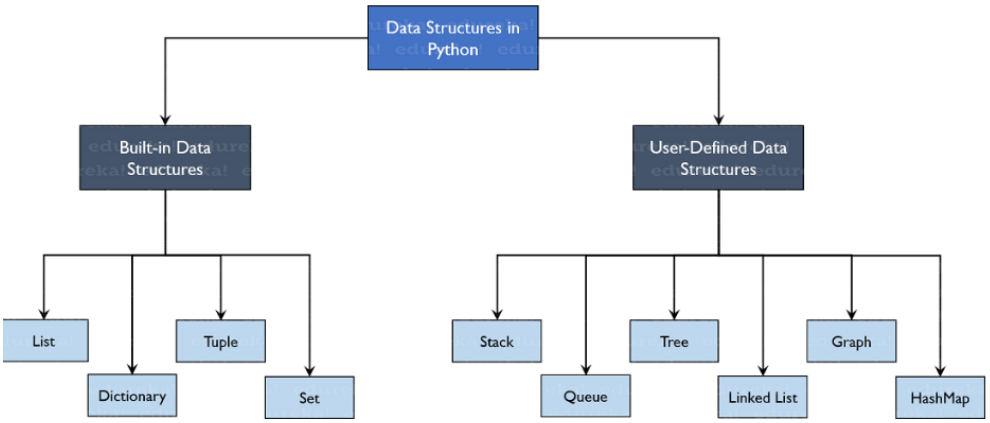
Telegram Group

Facebook

Download Andriod App

Data Structures in Python

Data Structures are a way of organizing data so that it can be accessed more efficiently depending upon the situation. Data Structures are fundamentals of any programming language around which a program is built. Python helps to learn the fundamental of these data structures in a simpler way as compared to other programming languages.



1 Built-In Data Structures

- Built-in data structures in Python can be divided into two broad categories: mutable and immutable.
- Mutable data structures are those which we can modify – for example, by adding, removing, or changing their elements. Python has three mutable data structures: lists, dictionaries, and sets.
- Immutable data structures, on the other hand, are those that we cannot modify after their creation. The only basic built-in immutable data structure in Python is a tuple.

1.1 List

- Lists are dynamic, ordered collections of elements that can hold items of various data types, including numbers, strings, and even other lists. Their defining feature is their ability to be modified, expanded, or reduced during program execution. This versatility makes lists immensely useful for scenarios where data needs to be managed flexibly.
- Lists are created using square brackets, and elements within them are separated by commas.
- Lists are mutable, ordered sequences of elements in Python. Declared using square brackets `[ ]`.
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- When we say that lists are ordered, it means that the items have a defined order, and that order will not change. If you add new items to a list, the new items will be placed at the end of the list.
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Operations:

1. **Creation:**

```
my_list = [1, 2, 3, 'a', 'b']
```

2. **Accessing Elements:**
   - Indexing: my_list[0] returns the first element.
   - Slicing: my_list[1:3] returns a sublist from index 1 to 2.
3. **Adding Elements:**
   - my_list.append(4) adds 4 to the end.
   - my_list.insert(1, 'x') inserts 'x' at index 1.
4. **Removing Elements:**
   - my_list.remove('a') removes the first occurrence of 'a'.
   - my_list.pop(2) removes the element at index 2.
5. **Other Operations:**
   - len(my_list) returns the length.
   - my_list + [5, 6] concatenates lists.

1.2 Tuples

- Tuples resemble lists in that they are ordered collections of elements, but they have a significant difference: they are immutable, meaning their content cannot be changed after creation.
- Tuples are defined using parentheses, and elements are separated by commas.
- Tuples are immutable, ordered sequences. Declared using parentheses `( )`.
- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Operations:

1. **Creation:**

```
my_tuple = (1, 2, 'a', 'b')
```

2. **Accessing Elements:** Similar to lists, using indexing and slicing.
3. **Immutable Nature:** Elements cannot be added or removed after creation.

1.3 Sets

- Set items are unordered, unchangeable, and do not allow duplicate values.
- Sets are unordered collections of unique elements. Declared using curly braces { } or `set()`.
- Set items are unordered, unchangeable, and do not allow duplicate values.
- Set items are unchangeable, meaning that we cannot change the items after the set has been created.
- Once a set is created, you cannot change its items, but you can remove items and add new items.
- Sets cannot have two items with the same value.

Operations:

1. **Creation:**

```
my_set = {1, 2, 3, 'a', 'b'}
```

2. **Adding and Removing Elements:**
   - my_set.add(4) adds 4 to the set.
   - my_set.remove('a') removes 'a'.
3. **Set Operations:**
   - Union: `set1 | set2`.
   - Intersection: `set1 & set2`.
   - Difference: `set1 - set2`.

1.4 Dictionaries

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are key-value pairs. Declared using curly braces { } with key-value pairs.
- Dictionary items are ordered, changeable, and does not allow duplicates.
- Dictionary items are presented in key:value pairs, and can be referred to by using the key name
- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

  Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

  As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Dictionaries cannot have two items with the same key

Operations:

1. **Creation:**

```
my_dict = {'key1': 'value1', 'key2': 2, 'key3': [1, 2, 3]}
```

2. **Accessing and Modifying Elements:**
   - my_dict['key1'] returns 'value1'.
   - my_dict['key2'] = 3 modifies the value of 'key2'.
3. **Dictionary Operations:**
   - my_dict.keys() returns keys.
   - my_dict.values() returns values.
   - my_dict.items() returns key-value pairs.

1.5 Comparison of List, Tuple, Set, and Dictionary in Python

1. **Homogeneity:**
   - **List:** Non-homogeneous type, can store various elements.
   - **Tuple:** Non-homogeneous type, stores various elements.
   - **Set:** Non-homogeneous type, stores various elements in a single row.
   - **Dictionary:** Non-homogeneous type, stores key-value pairs.
2. **Representation:**
   - **List:** Represented by [ ].
   - **Tuple:** Represented by ( ).
   - **Set:** Represented by { }.
   - **Dictionary:** Represented by { }.
3. **Duplicate Elements:**

- **List and Tuple:** Allow duplicate elements.
- **Set:** Does not allow duplicate elements.
- **Dictionary:** Keys are not duplicated.

4. **Nested Among All:**
   - **List:** Can be nested.
   - **Tuple:** Can be nested.
   - **Set:** Can be nested.
   - **Dictionary:** Can be nested.
5. **Example:**
   - **List:** [6, 7, 8, 9, 10].
   - **Tuple:** (6, 7, 8, 9, 10).
   - **Set:** {6, 7, 8, 9, 10}.
   - **Dictionary:** {6, 7, 8, 9, 10}.
6. **Function for Creation:**
   - **List:** `list()` function.
   - **Tuple:** `tuple()` function.
   - **Set:** `set()` function.
   - **Dictionary:** `dict()` function.
7. **Mutation:**
   - **List:** Mutable.
   - **Tuple:** Immutable.
   - **Set:** Mutable.
   - **Dictionary:** Mutable, but keys are not duplicated.
8. **Order:**
   - **List and Tuple:** Ordered.
   - **Set:** Unordered.
   - **Dictionary:** Ordered.
9. **Empty Elements:**
   - **List:** `l = []`.
   - **Tuple:** `t = ()`.
   - **Set:** `a = set()` or `b = set(a)`.
   - **Dictionary:** `d = {}`.

User-Defined Data Structures

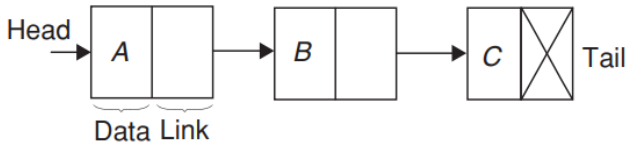
User-defined data structures are not inbuilt in Python, but we can still implement them. We can use the existing functional options in Python to create new data structures. For example, when we say a list = [], Python recognizes it as a list and calls everything related to a list. But when we say a linked list or a queue, Python won't know what these are.

2 Linked Lists

A **linked list** is a linear data structure where each element, called a **node**, contains data and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, allowing for efficient insertions and deletions.

Linked Lists in Python

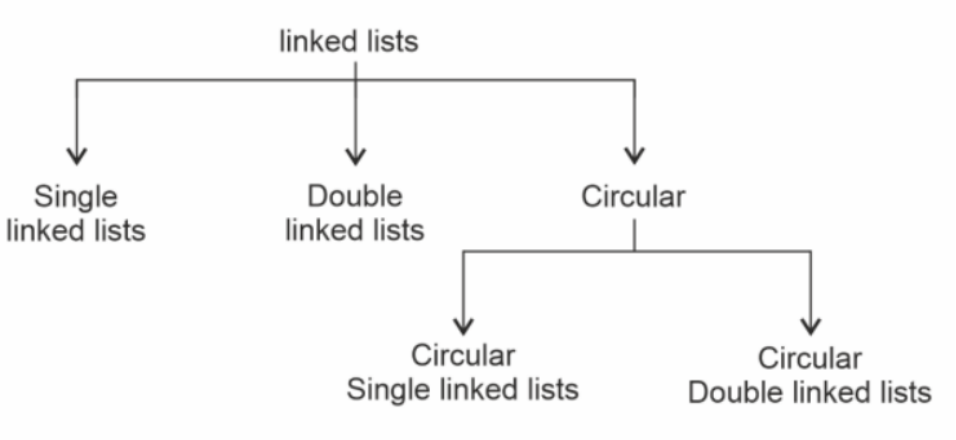
- A linked list is a linear data structure consisting of nodes where each node points to the next node in the sequence. It does not have a fixed size and is dynamic in nature.
- Generally, the node of the linked list is represented as a self-referential structure.
- The linked list elements are accessed with special pointer(s) called head and tail.
- A linked list node consists of two parts:
  - **Data:** Holds the value of the node.
  - **Next (and Previous for Doubly Linked List):** Points to the next (and previous) node in the sequence.



- The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk.
- Linked lists allow insertion and removal of nodes at any point in the list.
- Finding a node that contains a given data, or locating the place where a new node should be inserted may require scanning most or all of the list elements.
- The list element does not have to occupy contiguous memory.
- Adding, insertion or deletion of list elements can be accomplished with minimal disruption of neighbouring nodes

Important points about Linked lists:

- A linked list is an ordered collection of elements.
- A linked list is also used to implement other user-defined data structures like stack and queue.
- Using the collections module in Python, we can use the deque object to implement operations like insert and delete on linked lists.
- The first node in a linked list is the head, and we must start all the operations on the linked list from it.
- The last node of the linked list refers to None showing that the linked list is complete.



2.1 Types of Linked Lists:

2.1.1 Singly Linked List:

Each node points to the next node in the sequence.

2.1.2 Doubly Linked List:

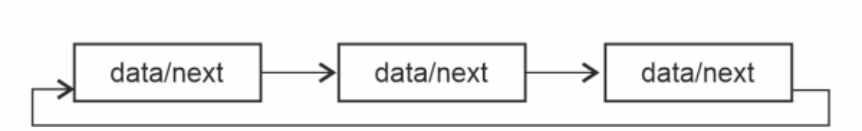
- In a double-linked list, every node will have three sections. Head holds the reference of the first node, the "previous" section of the first node holds None, and the next field of the last node refers to None.
- Each node will hold two references along with the data, one to its previous node and the next to the succeeding node.
- Each node points to both the next and the previous nodes.
- We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

- The operations which can be performed in SLL can also be preformed on DLL.
- The major difference is that we have to adjust double reference as compared to SLL.
- We can traverse or display the list elements in forward as well as in reverse direction.

2.1.3 Circular Single Linked Lists

The circular-linked list is completely the same as SLL, except, in CLL the last (Tail) node points to first (Head) node of list.

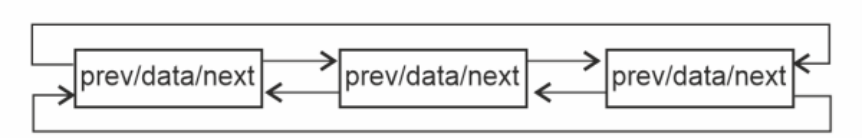
So, the Insertion and Deletion operation at Head and Tail are little different from SLL.



2.1.4 Circular Double Linked Lists

Double circular-linked list can be traversed in both directions again and again. DCL is very similar to DLL, except the last node's next pointer points to first node of list and first node's previous pointer points to last node of list.

So, the insertion and deletion operations at head and tail in DCL are little different in adjusting the reference as compared to DLL.



Basic Operations

- **Traversal**: Accessing each node in the list sequentially.
- **Insertion**: Adding a new node to the list.
- **Deletion**: Removing a node from the list.
- **Search**: Finding a node with a given value.

Advantages of Linked Lists

- Efficient insertions and deletions.
- Dynamic sizing.
- Memory utilization is efficient for unpredictable data sizes.

Disadvantages of Linked Lists

- No random access; nodes must be accessed sequentially.
- Extra memory is required for storing pointers.
- Traversal is slower compared to arrays.

2.2 Python Implementation of a Singly Linked List

Defining the Node Class

```
class Node:
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null
```

Defining the Linked List Class

```
class LinkedList:
    def __init__(self):
        self.head = None # Initialize head as null

    # Method to print the linked list
    def print_list(self):
        temp = self.head
        while temp:
            print(temp.data, end=' ')
            temp = temp.next
```

Insertion Operations

```
    # Function to insert a new node at the beginning
    def insert_at_beginning(self, new_data):
        new_node = Node(new_data) # Create a new node
        new_node.next = self.head # Link the new node to the head
        self.head = new_node # Make new node the head
```

```
    # Function to insert a new node at the end
    def insert_at_end(self, new_data):
        new_node = Node(new_data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
```

```
    # Function to insert a new node after a given node
    def insert_after(self, prev_node, new_data):
        if prev_node is None:
            print("Previous node must be in the LinkedList.")
            return
        new_node = Node(new_data)
        new_node.next = prev_node.next
        prev_node.next = new_node
```

Deletion Operations

```
    # Function to delete the first occurrence of a node by key
    def delete_node(self, key):
        temp = self.head

        # If head node holds the key
        if temp is not None:
            if temp.data == key:
                self.head = temp.next
                temp = None
                return

        # Search for the key
```

```
        while temp is not None:
            if temp.data == key:
                break
            prev = temp
            temp = temp.next

        # Key not present
        if temp is None:
            return

        # Unlink the node
        prev.next = temp.next
        temp = None
```

Example Usage

```
# Initialize the linked list
llist = LinkedList()

# Insert elements
llist.insert_at_end(1)
llist.insert_at_beginning(2)
llist.insert_at_beginning(3)
llist.insert_at_end(4)
llist.insert_after(llist.head.next, 5)

# Print the linked list
print("Linked List:")
llist.print_list() # Output: 3 2 5 1 4

# Delete a node
llist.delete_node(5)
print("\nAfter Deletion:")
llist.print_list() # Output: 3 2 1 4
```

2.3 Complexity Analysis

Inserting at the Beginning

Operation	Time Complexity
Traversal	$O(n)$
Insertion at Beginning	$O(1)$
Insertion at End	$O(n)$
Insertion after a Node	$O(1)$
Deletion	$O(n)$
Search	$O(n)$

Applications of Linked Lists

- Implementing stacks and queues.
- Managing dynamic memory allocation.
- Representation of graphs (adjacency lists).
- Implementation of hash tables (chaining method).

Advantages over Arrays

- Dynamic size adjustment.
- Efficient insertions and deletions without shifting elements.

Limitations Compared to Arrays

- No random access; accessing an element requires traversal from the head.
- Extra memory overhead due to storage of pointers.

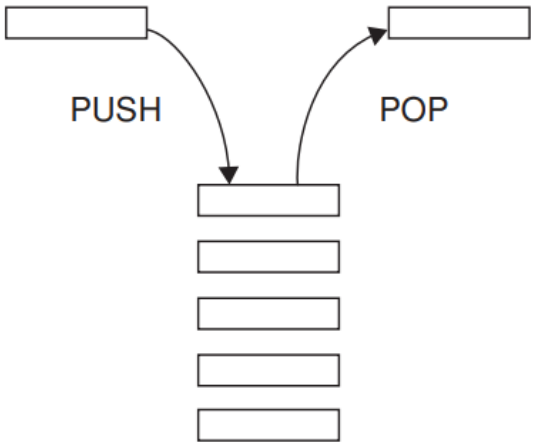
3 Stacks and Queues

Stacks and queues are fundamental linear data structures used for storing and managing collections of data elements. They differ in the order in which elements are added and removed.

3.1 Stacks

Definition

- A stack is a last in first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations, **PUSH and POP**
- The PUSH operation adds an item to the top of the stack, hiding any items already on the stack or initializing the stack if it is empty.
- The POP operation removes an item from the top of the stack, and returns the poped value to the caller.
- Elements are removed from the stack in the reverse order to the order of their insertion. Therefore, the lower elements are those that have been on the stack for the longest period.



The functions associated with stack are:

- empty() – Returns whether the stack is empty – Time Complexity: O(1)
- size() – Returns the size of the stack – Time Complexity: O(1)
- top() / peek() – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- push(a) – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- pop() – Deletes the topmost element of the stack – Time Complexity: O(1)

Implementation of Stack

Python offers various ways to implement the stack. In this section, we will discuss the implementation of the stack using Python and its module.

We can implement a stack in Python in the following ways.

- **Using List**

  Python list can be used as the stack. It uses the append() method to insert elements to the list where stack uses the push() method. The list also provides the pop() method to remove the last element, but there are shortcomings in the list. The list becomes slow as it grows.

  The list stores the new element in the next to other. If the list grows and out of a block of memory then Python allocates some memory.

- **Using collection.deque**

  The collection module provides the deque class, which is used to creating Python stacks. The deque is pronounced as the "deck" which means "double-ended queue". The deque can be preferred over the list because it performs append and pop operation faster than the list. The time complexity is O(1), where the list takes O(n).

- **Using queue module**

  The queue module has the LIFO queue, which is the same as the stack. Generally, the queue uses the put() method to add the data and the () method to take the data.

Characteristics of Stacks

- **LIFO Structure**: The most recently added element is accessed first.
- **Operations**:
  - **Push**: Add an element to the top of the stack.
  - **Pop**: Remove the top element from the stack.
  - **Peek/Top**: Retrieve the top element without removing it.
  - **IsEmpty**: Check if the stack is empty.

Applications of Stacks

- **Function Call Management**: Managing function calls in programming languages (call stack).
- **Expression Evaluation**: Parsing and evaluating expressions in compilers.
- **Undo Mechanisms**: Implementing undo operations in software applications.
- **Backtracking Algorithms**: Depth-first search in trees and graphs.

Python Implementation of a Stack

In Python, stacks can be implemented using lists or the `collections.deque` module.

```
class Stack:
    def __init__(self):
        self.stack = []

    # Push operation
    def push(self, item):
        self.stack.append(item)

    # Pop operation
    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            return None # Stack is empty

    # Peek operation
    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            return None
```

```
    # Check if stack is empty
    def is_empty(self):
        return len(self.stack) == 0

    # Display stack elements
    def display(self):
        print("Stack elements:", self.stack)
```

Example Usage

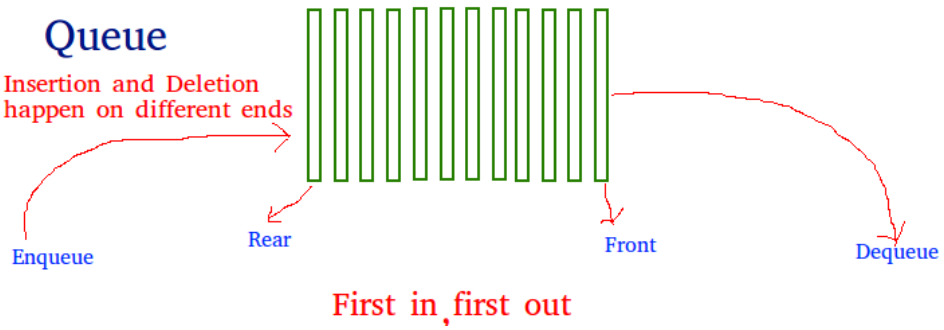
```
# Initialize the stack
my_stack = Stack()

# Push elements
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)

# Display stack
my_stack.display() # Output: Stack elements: [10, 20, 30]

# Pop element
print("Popped:", my_stack.pop()) # Output: Popped: 30

# Peek at top element
print("Top element:", my_stack.peek()) # Output: Top element: 20
```

Complexity Analysis of Stack Operations

Operation	Time Complexity
Push	$O(1)$
Pop	$O(1)$
Peek	$O(1)$
IsEmpty	$O(1)$

3.2 Queues

A queue is an ordered collection of items from which items may be deleted at one end (called the front of queue) and into which items may be inserted at the other end (called rear of queue). Queue is a linear data structure that maintains the data in first infirst out (FIFO) order



Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
- **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 – Time Complexity : O(1)
- **Front:** Get the front item from queue – Time Complexity : O(1)
- **Rear:** Get the last item from queue – Time Complexity : O(1)

Characteristics of Queues

- **FIFO Structure**: The earliest added element is accessed first.
- **Operations**:
  - **Enqueue**: Add an element to the rear of the queue.
  - **Dequeue**: Remove an element from the front of the queue.
  - **Front**: Retrieve the front element without removing it.
  - **IsEmpty**: Check if the queue is empty.

Applications of Queues

- **Scheduling**: Managing tasks in operating systems (CPU scheduling).
- **Buffering**: Handling data streams in networks and I/O systems.
- **Breadth-First Search**: Traversing trees and graphs.
- **Print Spooling**: Managing print jobs in a printer queue.

Python Implementation of a Queue

1. **Using list**

   List is a Python's built-in data structure that can be used as a queue. Instead of enqueue() and dequeue(), append() and pop() function is used. However, lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring O(n) time.

2. **Using collections.deque**

   Queue in Python can be implemented using deque class from the collections module. Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity. Instead of enqueue and deque, append() and popleft() functions are used.

3. **Using queue.Queue**

   Queue is built-in module of Python which is used to implement a queue. queue.Queue(maxsize) initializes a variable to a maximum size of maxsize. A maxsize of zero '0' means a infinite queue. This Queue follows FIFO rule.

In Python, queues can be implemented using lists, but for efficient queue operations, it is better to use `collections.deque` or the `queue` module.

```
from collections import deque

class Queue:
    def __init__(self):
        self.queue = deque()

    # Enqueue operation
    def enqueue(self, item):
        self.queue.append(item)

    # Dequeue operation
    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        else:
            return None # Queue is empty

    # Front operation
    def front(self):
        if not self.is_empty():
            return self.queue[0]
        else:
            return None

    # Check if queue is empty
    def is_empty(self):
        return len(self.queue) == 0

    # Display queue elements
    def display(self):
        print("Queue elements:", list(self.queue))
```

```
# Initialize the queue
my_queue = Queue()

# Enqueue elements
my_queue.enqueue(10)
my_queue.enqueue(20)
my_queue.enqueue(30)
```

```
# Display queue
my_queue.display() # Output: Queue elements: [10, 20, 30]

# Dequeue element
print("Dequeued:", my_queue.dequeue()) # Output: Dequeued: 10

# Front element
print("Front element:", my_queue.front()) # Output: Front element: 20
```

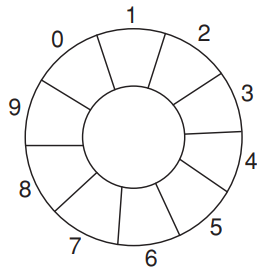
Complexity Analysis of Queue Operations

Example Usage

Operation	Time Complexity
Enqueue	$O(1)$
Dequeue	$O(1)$
Front	$O(1)$
IsEmpty	$O(1)$

3.2.1 Circular Queue

As the items from a queue get deleted, the space for that item is reclaimed. Those queue positions continue to be empty. This problem is solved by circular queues. Instead of using a linear approach, a circular queue takes a circular approach; this is why a circular queue does not have a beginning or end.



The advantage of using a circular queue over linear queue is efficient usage of memory.

3.2.2 Priority Queue

- In priority queue, the intrinsic ordering of elements does determine the results of its basic operations. There are two types of priority queues.
- Ascending priority queue is a collection of items in which items can be inserted arbitrarily and from which only the smallest items can be removed.
- Descending priority queue is similar but allows deletion of the largest item.

3.3 Implementing Stacks and Queues Using Linked Lists

Stacks and queues can also be implemented using linked lists to overcome size limitations.

Stack Implementation Using a Linked List

- The `head` of the linked list represents the top of the stack.
- **Push**: Insertion at the beginning.
- **Pop**: Deletion from the beginning.

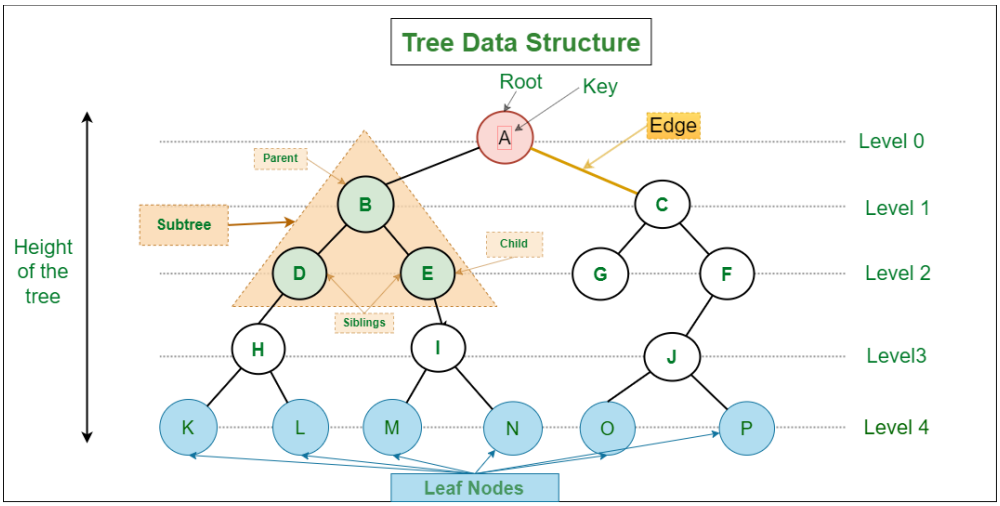
Queue Implementation Using a Linked List

- The `head` represents the front, and the `tail` represents the rear of the queue.
- **Enqueue**: Insertion at the end.
- **Dequeue**: Deletion from the beginning.

4 Trees

- Tree is non-linear data structure designated at a special node called root and elements are arranged in levels without containing cycles.
  (or)
  The tree is
  1. Rooted at one vertex
  2. Contains no cycles
  3. There is a sequence of edges from any vertex to any other
  4. Any number of elements may connect to any node (including root)
  5. A unique path traverses from root to any node of tree
  6. Tree stores data in hierarchical manner
  7. The elements are arranged in layers

- A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
- The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.
- This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. B is the parent node of D, E.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: D, E are the child nodes of B.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. A is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. K, L, M, N, O, P, G are the leaf nodes of the tree.

- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. A,B are the ancestor nodes of the node E
- **Descendant:** Any successor node on the path from the leaf node to that node. E,I are the descendants of the node B.
- **Sibling:** Children of the same parent node are called siblings. D,E are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node. Subtree: Any node of the tree along with its descendant.

4.1 Binary Trees

A **binary tree** is a hierarchical data structure in which each node has at most two children, referred to as the **left child** and the **right child**. It is a special case of an n-ary tree where $n = 2$.

Terminology

- **Node**: The basic unit of a binary tree containing data and references to its children.
- **Root**: The topmost node of the tree.
- **Leaf**: A node with no children.
- **Internal Node**: A node with at least one child.
- **Height**: The length of the longest path from the root to a leaf.
- **Depth**: The length of the path from the root to a node.
- **Subtree**: A tree consisting of a node and its descendants.

Types of Binary Trees

1. **Full Binary Tree**: Every node has 0 or 2 children.
2. **Perfect Binary Tree**: All internal nodes have two children, and all leaves are at the same level.
3. **Complete Binary Tree**: All levels are completely filled except possibly the last, which is filled from left to right.
4. **Balanced Binary Tree**: The height of the tree is minimized, leading to operations being performed efficiently.

Properties of Binary Trees

For a binary tree of height h:

- Each subtree is a binary tree.
- Degree of any node is 0/1/2.
- The maximum number of nodes in a tree with height h is $2^{h+1} - 1$.
- The maximum number of nodes at level i is 2^{i-1}.
- For any non-empty binary tree, the number of terminal nodes with n_2, nodes of degree 2 is $N_0 = n_2 + 1$.
- The maximum number of nodes in a tree with depth d is $2^d - 1$.

4.2 Binary Search Trees (BST)

A **binary search tree** is a binary tree in which every node satisfies the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Properties of BSTs

- **Inorder Traversal**: Yields the nodes in ascending order.
- **Search Efficiency**: Average-case time complexity is $O(\log n)$.
- **Insertion and Deletion**: Maintains the BST property after operations.

Operations on BSTs

Search Operation

Algorithm:

1. Start at the root node.
2. If the key matches the root's key, return the node.
3. If the key is less than the root's key, search the left subtree.
4. If the key is greater than the root's key, search the right subtree.
5. Repeat until the key is found or the subtree is null.

```
def search(root, key):
    if root is None or root.val == key:
        return root
    if key < root.val:
        return search(root.left, key)
    else:
        return search(root.right, key)
```

Insertion Operation
Python Implementation:

Algorithm:

1. Start at the root node.
2. If the tree is empty, create a new node as the root.
3. If the key is less than the current node's key, go to the left subtree.
4. If the key is greater than the current node's key, go to the right subtree.
5. Repeat until the correct position is found and insert the new node.

```
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
```

Deletion Operation
Python Implementation: Deleting a node from a BST has three cases:

1. **Node with no children (Leaf Node)**: Remove the node.
2. **Node with one child**: Replace the node with its child.
3. **Node with two children**:
    - Find the inorder successor (smallest value in the right subtree) or inorder predecessor (largest value in the left subtree).
    - Replace the node's value with the successor's value.
    - Delete the successor node.

Time Complexity of BST Operations

Operation	Average Case	Worst Case
Search	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$

4.3 Tree Traversal Methods

Traversal refers to the process of visiting each node in a tree data structure exactly once in some order.

Depth-First Traversals

1. **Inorder Traversal (Left, Root, Right)**
2. **Preorder Traversal (Root, Left, Right)**
3. **Postorder Traversal (Left, Right, Root)**

4.3.1 Example Usage

```
# Constructing the following binary tree
# 1
# / \
# 2 3
# / \ / \
# 4 5 6 7

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

```
root.right.left = Node(6)
root.right.right = Node(7)

print("Inorder traversal:")
inorder(root) # Output: 4 2 5 1 6 3 7

print("\nPreorder traversal:")
preorder(root) # Output: 1 2 4 5 3 6 7

print("\nPostorder traversal:")
postorder(root) # Output: 4 5 2 6 7 3 1

print("\nLevel-order traversal:")
level_order(root) # Output: 1 2 3 4 5 6 7
```

4.4 Balancing Binary Search Trees

Unbalanced BSTs can degenerate into linked lists with $O(n)$ time complexity. To prevent this, self-balancing BSTs are used:

- **AVL Trees**: Maintain balance by ensuring the heights of left and right subtrees differ by at most one.
- **Red-Black Trees**: Use color properties to maintain balance during insertions and deletions.

Applications of Binary Search Trees

- **Searching and Sorting**: Efficiently search and sort data.
- **Database Indexing**: Implement indexes for quick data retrieval.
- **Symbol Tables**: Used in compilers and interpreters for variable/function lookup.

5 Heaps (Optional for GATE DA)

A **heap** is a specialized tree-based data structure that satisfies the **heap property**. In a **max heap**, for any given node C, if P is a parent node of C, then the key (value) of P is greater than or equal to the key of C. In a **min heap**, the key of P is less than or equal to the key of C.

Types of Heaps

1. **Max Heap**: The value of each node is less than or equal to the value of its parent, with the highest value at the root.
2. **Min Heap**: The value of each node is greater than or equal to the value of its parent, with the lowest value at the root.

Properties of Heaps

- **Complete Binary Tree**: A heap is always a complete binary tree; all levels are completely filled except possibly the last level, which is filled from left to right.
- **Heap Property**: The key at the root must be either minimum or maximum among all keys present in the binary heap.

Heap Operations

1. **Insertion**:
   - Insert the new key at the end of the heap (last level).
   - Heapify the tree by comparing the inserted key with its parent and swap if necessary.
   - Continue until the heap property is restored.
2. **Deletion (Extract Max/Min)**:
   - Replace the root key with the last key in the heap.
   - Remove the last key.
   - Heapify the root node by comparing it with its children and swap if necessary.
   - Continue until the heap property is restored.
3. **Heapify**:
   - A process to maintain the heap property by moving a node down the tree (sift-down) or up the tree (sift-up).

Applications of Heaps

- **Priority Queues**: Implementing priority queues where highest (or lowest) priority element is accessed first.
- **Heap Sort**: An efficient comparison-based sorting algorithm.
- **Graph Algorithms**: Used in algorithms like Dijkstra's shortest path and Prim's minimum spanning tree.
- **Median Maintenance**: Finding median in a stream of integers.

Time Complexity of Heap Operations

Operation	Time Complexity
Insertion	$O(\log n)$
Extraction (Delete)	$O(\log n)$
Find Max/Min	$O(1)$
Build Heap	$O(n)$

5.1 Heap Sort Algorithm

1. Build a max heap from the input data.
2. Repeat the following steps until the heap size is greater than 1:
    - Swap the first element (maximum) with the last element.
    - Reduce the heap size by one.
    - Heapify the root element to maintain the heap property.

Python Implementation of Heap Sort

```
def heap_sort(arr):
    n = len(arr)

    # Build a max heap
    for i in range(n//2 -1, -1, -1):
        max_heapify(arr, n, i)

    # Extract elements one by one
    for i in range(n-1, 0, -1):
        # Swap
        arr[i], arr[0] = arr[0], arr[i]
        # Heapify root element
        max_heapify(arr, i, 0)

def max_heapify(arr, n, i):
    largest = i
    left = 2*i + 1
    right = 2*i +2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, n, largest)

# Example usage
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print("Sorted array:", arr) # Output: [5, 6, 7, 11, 12, 13]
```

6 Hashing

Hashing is a technique used to uniquely identify a specific object from a group of similar objects. Hashing is implemented using a data structure called a **hash table**, which stores key-value pairs and uses a hash function to compute an index into an array of buckets from which the desired value can be found.

Hash Functions

A **hash function** is a function that takes input (or 'message') and returns a fixed-size string of bytes. The output is called the hash value or hash code.

Properties of a Good Hash Function

- **Deterministic**: Same key always produces the same hash value.
- **Efficiently Computable**: Hash function should be easy to compute.
- **Uniform Distribution**: Should distribute keys uniformly across the table.
- **Minimize Collisions**: Should minimize the number of collisions.

Collision Resolution Techniques

Since hash functions can map multiple keys to the same hash value, collision resolution is necessary.

Open Addressing

- **Linear Probing**: Search sequentially for the next available slot.
- **Quadratic Probing**: Use quadratic function to find next slot.
- **Double Hashing**: Use a second hash function to calculate the offset.

Chaining

- Store a linked list of all elements that hash to the same slot.

Applications of Hashing

- **Data Storage**: Implementing associative arrays and database indexing.
- **Caching**: Quick data retrieval in cache mechanisms.
- **Cryptography**: Hash functions are used in cryptographic algorithms.
- **Checksum Algorithms**: Data integrity verification.

6.1 Time Complexity of Hash Table Operations

Operation	Average Case
Insertion	$O(1)$
Search	$O(1)$
Deletion	$O(1)$

Note: In the worst-case scenario (when all keys collide), the time complexity can degrade to $O(n)$.

6.2 Load Factor

The **load factor** α is defined as:

$$\alpha = \frac{\text{Number of keys}}{\text{Size of the hash table}}$$

A higher load factor increases the chances of collisions. It's important to keep the load factor below a certain threshold (commonly 0.7) to maintain efficiency.

7 Searching Algorithms

Searching algorithms are designed to retrieve an element from any data structure where it is stored. The most common searching algorithms are:

7.1 Linear Search

Definition

Linear search, also known as sequential search, is the simplest searching algorithm. It checks each element of the list sequentially until a match is found or the whole list has been searched.

Algorithm

1. Start from the first element of the array.
2. Compare the current element with the target value.
3. If the current element matches the target, return its index.
4. If not, move to the next element.
5. Repeat steps 2-4 until the element is found or the array ends.

Python Implementation

```
def linear_search(arr, target):
    for index, element in enumerate(arr):
        if element == target:
            return index # Element found
    return -1 # Element not found

# Example usage
array = [5, 3, 8, 4, 2]
index = linear_search(array, 4)
print("Element found at index:", index) # Output: 3
```

Time and Space Complexity

- **Time Complexity**:
  - Best Case: $O(1)$ (Element found at the first position)
  - Average and Worst Case: $O(n)$
- **Space Complexity**: $O(1)$

7.2 Binary Search

- A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element. We can find the element's index position very fast using the binary search algorithm.
- There are many searching algorithms but the binary search is most popular among them.
- The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.
- Binary search is an efficient search algorithm that finds the position of a target value within a sorted array.
- In the binary search algorithm, we can find the element position using the following methods:
  1. **Recursive Method**
  2. **Iterative Method**

- The divide and conquer approach technique is followed by the recursive method. In this method, a function is called itself again and again until it found an element in the list.
- A set of statements is repeated multiple times to find an element's index position in the iterative method. The while loop is used for accomplish this task.
- Binary search is more effective than linear search because we don't need to search each list index. The list must be sorted to achieve the binary search algorithm.

Algorithm:

1. Start with the entire sorted array.
2. Set the lower bound to the first index and the upper bound to the last index.
3. Compute the middle index: `mid = (lower + upper) // 2`.
4. If the middle element is equal to the target, return its index.
5. If the target is less than the middle element, search in the left half. Otherwise, search in the right half.
6. Repeat the process until the target is found or the bounds overlap.

Python Implementation:

```
def binary_search(arr, target):
    lower, upper = 0, len(arr) - 1
    while lower <= upper:
        mid = (lower + upper) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            lower = mid + 1
        else:
            upper = mid - 1
    return -1
```

Example:

```
sorted_list = [1, 2, 4, 6, 7, 9]
target_value = 7
result = binary_search(sorted_list, target_value)
print(f"Index␣of␣{target_value}:␣{result}")
```

Time and Space Complexity

- **Time Complexity**:
  - Best Case: $O(1)$
  - Average and Worst Case: $O(\log n)$
- **Space Complexity**: $O(1)$ (Iterative implementation)

7.3 Comparison: Linear Search vs. Binary Search

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 preferable for small-sized data sets.	It is preferable for 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.

8 Basic Sorting Algorithms

A sorting algorithm is a method for arranging the elements of a list or collection in a specific order. Sorting is a fundamental operation in computer science and is applied to various tasks, such as searching, data analysis, and optimization. Sorting algorithms play a crucial role in organizing data efficiently, and there are various algorithms designed to achieve this task. These algorithms differ in their approaches, time complexity, and space complexity.

Purpose of sorting

- Sorting is a technique which reduces problem complexity and search complexity.
- Insertion sort takes $O(n^2)$ time in the worst case. It is a fast inplace sorting algorithm for small input sizes.
- Merge sort has a better asymptotic running time $O(nlogn)$, but it does not operate in inplace.
- Heap sort, sorts 'n' numbers inplace in $O(nlogn)$ time, it uses a data structure called heap, with which we can also implement a priority queue.
- Quick sort also sorts 'n' numbers in place, but its worst – case running time is $O(n^2)$. Its average case is $O(nlogn)$. The constant factor in quick sort's running time is small, This algorithm performs better for large input arrays.
- Insertion sort, merge sort, heap sort, and quick sort are all comparison-based sorts; they determine the sorted order of an input array by comparing elements

8.1 Selection Sort

- Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists.
- Selection Sort is another simple sorting algorithm. It divides the input list into a sorted and an unsorted region. The algorithm repeatedly selects the smallest (or largest, depending on the order) element from the unsorted region and swaps it with the first element of the unsorted region. The process is repeated until the entire list is sorted.
- **Time complexity:** $O(n^2)$ in all cases (worst, average, and best)
- **Space complexity:** $O(1)$
- **Basic idea:** Find the minimum element in the unsorted portion of the array and swap it with the first unsorted element. Repeat until the array is fully sorted.
- **Advantages:** Simple implementation, works well for small datasets, requires only constant space, in-place sorting algorithm
- **Disadvantages:** Inefficient for large datasets, worst-case time complexity of $O(n^2)$, not optimal for partially sorted datasets, not a stable sorting algorithm

Algorithm:

1. Find the minimum element in the unsorted part of the array.
2. Swap it with the first element of the unsorted part.
3. Move the boundary between the sorted and unsorted parts.
4. Repeat until the entire array is sorted.

Python Implementation:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

8.2 Bubble Sort

- Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements 'bubble' to the top of the list.
- **Time complexity:** $O(n^2)$ in the worst and average cases, $O(n)$ in the best case (when the input array is already sorted)
- **Space complexity:** $O(1)$
- **Basic idea:** Iterate through the array repeatedly, comparing adjacent pairs of elements and swapping them if they are in the wrong order. Repeat until the array is fully sorted.
- **Advantages:** Simple implementation, works well for small datasets, requires only constant space, stable sorting algorithm
- **Disadvantages:** Inefficient for large datasets, worst-case time complexity of $O(n^2)$, not optimal for partially sorted datasets

Algorithm:

1. Compare each pair of adjacent elements in the array.
2. Swap them if they are in the wrong order.
3. Repeat until no more swaps are needed.

Python Implementation:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

8.3 Insertion Sort

- Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms, such as quicksort, heapsort, or merge sort. However, it performs well for small datasets or partially sorted datasets.
- **Time complexity:** $O(n^2)$ in the worst and average cases, $O(n)$ in the best case (when the input array is already sorted)
- **Space complexity:** $O(1)$

- **Basic idea:** Build up a sorted subarray from left to right by inserting each new element into its correct position in the subarray. Repeat until the array is fully sorted.
- **Advantages:** Simple implementation, works well for small datasets, requires only constant space, efficient for partially sorted datasets, stable sorting algorithm
- **Disadvantages:** Inefficient for large datasets, worst-case time complexity of $O(n^2)$

Algorithm:

1. Build a sorted array one element at a time.
2. Take each element and insert it into its correct position in the sorted array.

Python Implementation:

```
def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are greater than key,
        # to one position ahead of their current position
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Example usage
arr = [4, 3, 2, 1, 5]
insertion_sort(arr)
print("Sorted␣array␣is:", arr)
```

9 Divide and Conquer Algorithms: Mergesort and Quicksort

- Divide-and-conquer is a top down technique for designing algorithms that consists of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to fi nd and then composing the partial solutions into the solution of the original problem.
- Divide-and-conquer paradigm consists of following major phases:
  - Breaking the problem into several sub-problems that are similar to the original problem but smaller in size.
  - Solve the sub-problem recursively (successively and independently)
  - Finally, combine these solutions to sub-problems to create a solution to the original problem
- Divide and conquer is a powerful algorithmic paradigm that involves breaking a problem into subproblems, solving them independently, and combining their solutions to solve the original problem.

  This technique can be divided into the following three parts:
  Divide: This involves dividing the problem into smaller sub-problems.
  Conquer: Solve sub-problems by calling recursively until solved.
  Combine: Combine the sub-problems to get the final solution of the whole problem.

Divide-and-Conquer Examples

- Sorting: Merge sort and quick sort
- Binary tree traversals
- Binary Search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and Convex-hull algorithm

9.1 Mergesort

Merge Sort is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.

Algorithm:

1. **Divide:** Divide the unsorted list into n sublists, each containing one element.
2. **Conquer:** Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining.
3. **Combine:** The final result is a sorted list.

Python Implementation:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
```

```
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

9.2 Quicksort

Quicksort is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.

Algorithm:

1. **Divide:** Choose a pivot element and partition the array into two subarrays such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right.
2. **Conquer:** Recursively apply quicksort to the subarrays.
3. **Combine:** The final result is a sorted array.

Python Implementation:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```

9.3 Comparison of Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

10 Previous Year Questions of GATE DA 2024

1. Consider sorting the following array of integers in ascending order using an in-place Quicksort algorithm that uses the last element as the pivot:

60	70	80	90	100

The minimum number of swaps performed during this Quicksort is ____.
Answer: 0

2. The fundamental operations in a double-ended queue D are:

- `insertFirst(e)` – Insert a new element e at the beginning of D.
- `insertLast(e)` – Insert a new element e at the end of D.
- `removeFirst()` – Remove and return the first element of D.
- `removeLast()` – Remove and return the last element of D.

In an empty double-ended queue, the following operations are performed:

```
insertFirst(10)
insertLast(32)
a ← removeFirst()
insertLast(28)
insertLast(17)
a ← removeFirst()
a ← removeLast()
```

The value of a is ____.
Answer: 17

3. Let $F(n)$ denote the maximum number of comparisons made while searching for an entry in a sorted array of size n using binary search.

Which **ONE** of the following options is **TRUE**?

(a) $F(n) = F(\lfloor n/2 \rfloor) + 1$
(b) $F(n) = F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$
(c) $F(n) = F(\lfloor n/2 \rfloor)$
(d) $F(n) = F(n-1) + 1$

Answer: a

4. Consider the following Python function:

```
def fun(D, s1, s2):
    if s1 < s2:
        D[s1], D[s2] = D[s2], D[s1]
        fun(D, s1+1, s2-1)
```

What does this Python function `fun()` do? Select the **ONE** appropriate option below.

(a) It finds the smallest element in D from index $s1$ to $s2$, both inclusive.
(b) It performs a merge sort in-place on this list D between indices $s1$ and $s2$, both inclusive.
(c) It reverses the list D between indices $s1$ and $s2$, both inclusive.
(d) It swaps the elements in D at indices $s1$ and $s2$, and leaves the remaining elements unchanged.

Answer: c

5. Consider the following sorting algorithms:

(a) Bubble Sort

(b) Insertion Sort

(c) Selection Sort

Which **ONE** among the following choices of sorting algorithms sorts the numbers in the array [4, 3, 2, 1, 5] in increasing order after exactly **two passes** over the array?

(a) (i) only

(b) (iii) only

(c) (i) and (iii) only

(d) (ii) and (iii) only

Answer: b

6. Let H, I, L, and N represent height, number of internal nodes, number of leaf nodes, and the total number of nodes respectively in a rooted binary tree.

   Which of the following statements is/are always **TRUE**?

   (a) $L \leq I + 1$

   (b) $H + 1 \leq N \leq 2^{H+1} - 1$

   (c) $H \leq I \leq 2^H - 1$

   (d) $H \leq L \leq 2^{H-1}$

   Answer: a, b, c

7. Consider the following tree traversals on a full binary tree:

   (a) Preorder

   (b) Inorder

   (c) Postorder

   Which of the following traversal options is/are sufficient to uniquely reconstruct the full binary tree?

   (a) (i) and (ii)

   (b) (ii) and (iii)

   (c) (i) and (iii)

   (d) (ii) only

   Answer: a , b, c