Chapter 2: Review of Abstract Data Types

By: Ashok Basnet

Course Outline

1. Stacks
2. Queues
3. Priority Queues
4. Binary Trees
5. Dictionaries
6. Sets and Disjoint Set Union

Data Types

- **Two important things about data types:**
  - Defines a certain domain of values.
  - Defines Operations allowed on those values.
- **Example: int type**
  - Takes only integer values
  - Operations: **addition, subtraction, multiplication, bitwise operations** etc.
- We cannot perform **bitwise** and **%** operations in float.

User Defined Data Types

- In contrast to **primitive data types** (int, float, char), there is a concept of a **user defined data types.**
- The operations and values of user defined data types are not specified in the language itself but is specified by the user.
- **Example**: ***Structure in C*** -> By using structures, we are defining our own type by combining other data types.
- struct student {

  int id;

  char name[50];

  }

Abstract Data Types (ADT)

- ADTs are like ***user defined data types*** which defines operations on values using functions *without specifying what is there inside the function* and how the operation are performed.
- An ADT specifies *what operations can be performed* on a particular data structure, but it *does not prescribe how these operations should be implemented.*
- **Example:** Stack ADT
- A stack consists of elements of same type arranged in a sequential order.

Abstract Data Types (ADT)

- The idea behind ADTs is ***to separate the logical view*** of a data structure from its implementation details.
- This separation ***allows*** programmers to focus on the *functionality and behavior of the data structure without worrying about the specific implementation*.
- As a result, ADTs provide a way to ***abstract away*** the underlying complexities of data structures and promote modularity and code reusability.

Examples of ADT

- **Stack:** A data structure that follows the *Last In, First Out (LIFO) principle*, where elements are added and removed from the same end (top).
- **Queue:** A data structure that follows the *First In, First Out (FIFO) principle,* where elements are added at the rear and removed from the front.
- **List:** A collection of elements where *each element has a position or index*, and operations can include adding, removing, or accessing elements at a specific position.

Examples of ADT

- **Set:** A collection of *unique elements* with operations like adding, removing, and checking for the presence of an element.
- **Map (or Dictionary):** A collection of *key-value pairs*, where each key is associated with a value. Operations include inserting, deleting, and looking up values based on their keys.
- **Graph:** A collection of *nodes and edges*, where nodes represent entities, and edges represent relationships between entities.

Abstract Data Types (ADT)

- ***Initialization/Creation:***
  - Description: Initializes a new instance of the ADT.
  - Purpose: Creates a new instance of the data structure.
- ***Insertion/Adding:***
  - Description: Adds a new element to the data structure.
  - Purpose: Increases the size of the data structure by including a new element.
- ***Deletion/Removing:***
  - Description: Removes an element from the data structure.
  - Purpose: Decreases the size of the data structure by eliminating an element.

Abstract Data Types (ADT)

- **Access/Retrieval:**
  - Description: Retrieves the value of a specific element in the data structure.
  - Purpose: Obtains the value of an element without modifying the data structure.
- **Search:**
  - Description: Determines whether a particular element is present in the data structure.
  - Purpose: Checks for the existence of a specific element.
- **Traversal:**
  - Description: Visits and processes each element in the data structure.
  - Purpose: Iterates through all elements in the data structure.

Abstract Data Types (ADT)

- **Update/Modification:**
  - Description: Modifies the value of an existing element in the data structure.
  - Purpose: Changes the content of an element without adding or removing it.
- **Size/Length:**
  - Description: Returns the number of elements in the data structure.
  - Purpose: Provides information about the size or length of the data structure.

Abstract Data Types (ADT)

- **Emptiness Check:**
  - Description: Determines whether the data structure is empty.
  - Purpose: Indicates whether the data structure contains any elements.
- **Clear:**
  - Description: Removes all elements from the data structure.
  - Purpose: Resets the data structure to an empty state.
- The *class provides the implementation details*, *while the interface defines the set of operations that can be performed on the ADT*.

Abstract Data Types (ADT)

- Think of ADT as a black box which hides the ***inner structure and design of the data type*** from the user.
- There are multiple ways to implement an ADT.
- **Example:** A stack ADT can be implemented using ***arrays or linked lists.***
- The program which uses data structure is called **a client program**. Client program has access to the ADT i.e. interface.
- The program which implements the data structure is known as **the implementation**.

Stacks

- A stack is a ***linear data structure*** in which ***insertions and deletions*** are allowed only at the end, called the top of the stack.
- When we define a stack as an **ADT (or Abstract Data Type),** then we are only interested in knowing the stack operations from the ***user point of view.***
- Means we are not interested in knowing the implementation details at this moment.
- We are only interested in knowing what ***types of operations*** we can perform on stack.

Why Abstract Data Types?

- Let say, if someone wants to use the **stack** in the program, then he can simply use **push** and **pop** operations without knowing its implementation.
- Also, if in future, the implementation of stack is changed from **array** to **linked list**, then the client program will work in the same way without being affected.
- Hence, ADT provides Abstraction.

Primary Stack Operations

- **push (data) :** Inserts data onto stack.
- **pop ( ) :** Deletes the last inserted element from the stack
- **top ( ) :** returns the last inserted element without removing it.
- **size ( ) :** returns the size or the number of elements in the stack.
- **isEmpty ( ) :** returns True if the stack is empty, else returns False.
- **isFull ( ) :** returns True if the stack is full, else returns False.

Push and Pop Operation

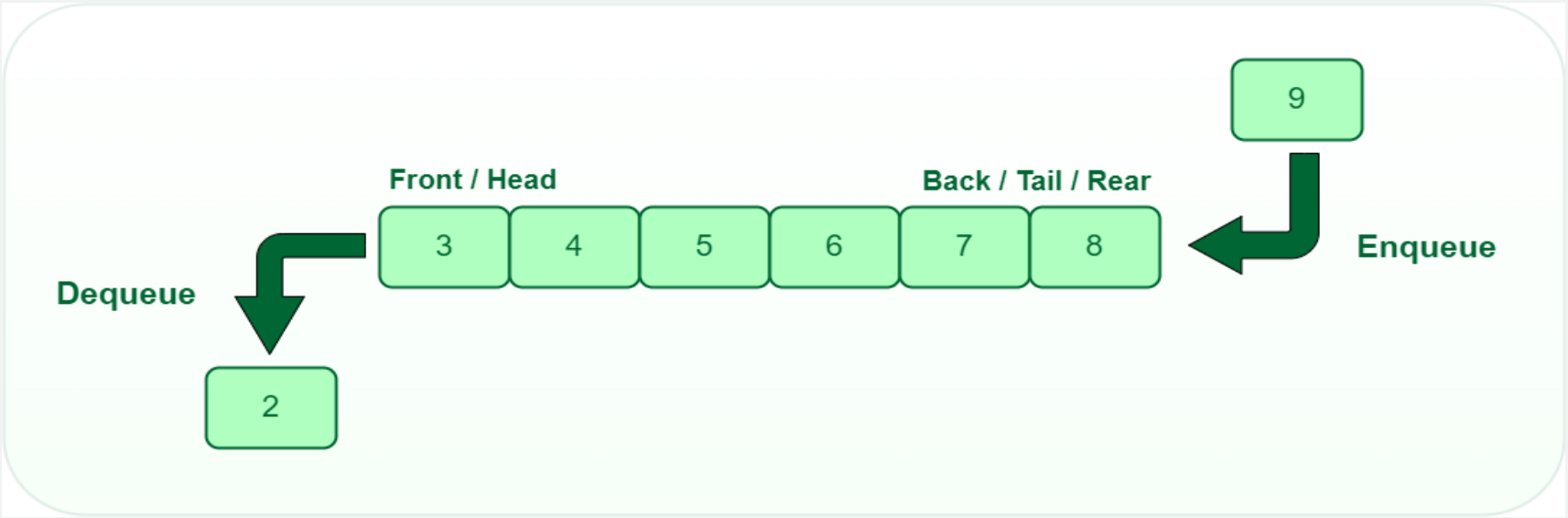
- For the **push** operation:
  - Top is incremented by 1.
  - New element is pushed at the position top

- For the **pop** operation:
  - The element at the position of top is deleted.
  - Top is decremented by 1

Pop an element

- How to delete the element at index 3?
- We cannot simply ***remove*** the array element.
- Still, we can give the user an illusion that the array element is deleted by ***decrementing*** the top variable.
- Top variable always ***keeps track*** of the topmost element of the stack.
- If the top is decremented by 1 then the user will perceive that the topmost element is deleted.

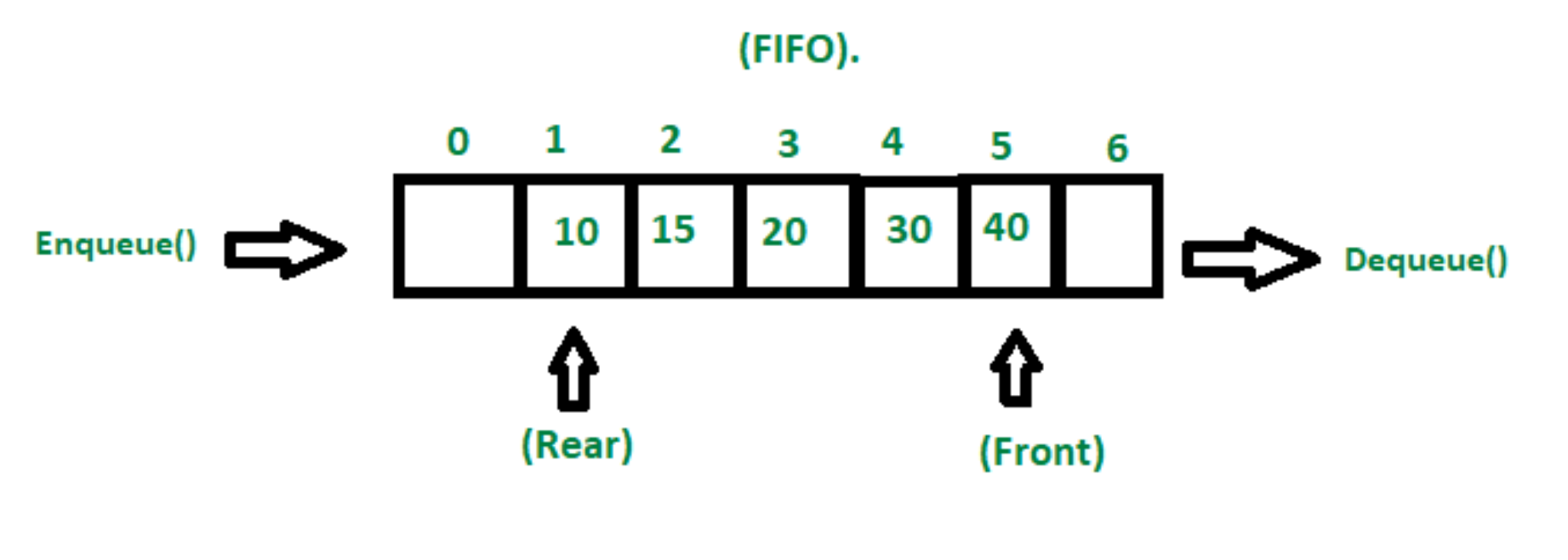
Queues



- A queue is defined as ***a linear data structure*** that is open at both ends and the operations are performed in First In First Out (FIFO) order.
- We define a queue to be a list in which *all additions to the list are made at one end, and all deletions from the list are made at the other end.*
- The element which is ***first pushed*** into the order, the ***operation is first performed*** on that.

FIFO Principle of Queue:

- A ***Queue*** is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).



Queue Representation

Like stacks, Queues can also be represented in an array:

In this representation, the Queue is implemented using the array. Variables used in this case are

- **Queue:** the name of the array storing queue elements.
- **Front**: the index where the first element is stored in the array representing the queue.
- **Rear:** the index where the last element is stored in an array representing the queue.

Array implementation of queue

To implement a queue using an array,

- **create an array**: array of size n and
- take two variables **front and rear both** of which will be initialized to 0 which means the queue is currently empty.
- Element
  - **rear** is the index up to which the elements are stored in the array and
  - **front** is the index of the first element of the array.

Enqueue

Addition of an element to the queue.

Adding an element will be performed after checking whether the queue is full or not.

If rear < n which indicates that the array is not full then store the element at arr[rear] and increment rear by 1

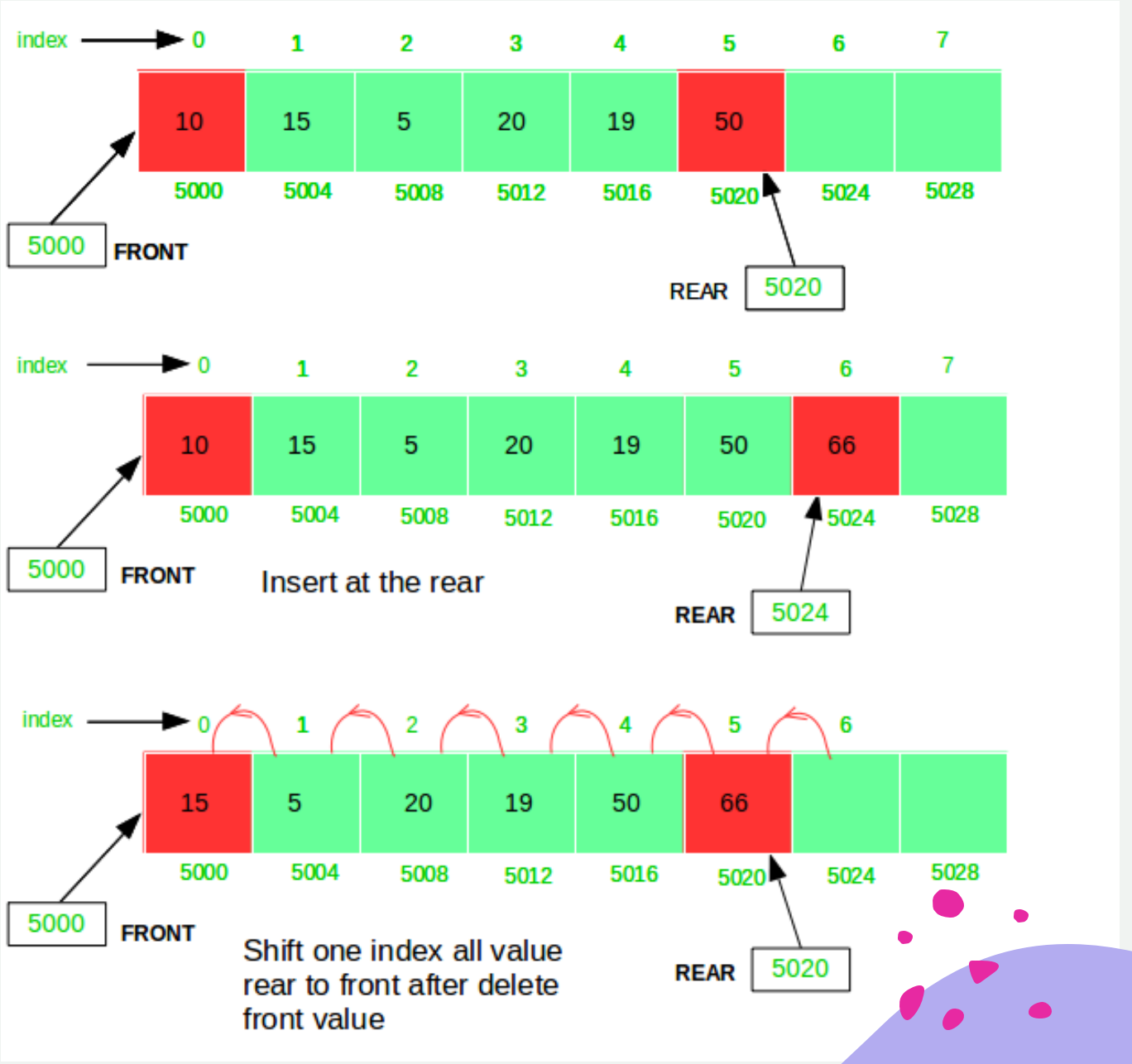
But if rear == n then it is said to be an Overflow condition as the array is full.

Dequeue

- Removal of an element from the queue.
- An element can only be ***deleted*** when there is at least an element to delete i.e. rear > 0.
- Now, the element at **array[front]** can be deleted but all the remaining elements have to shift to the left by one position in order for the dequeue operation ***to delete the second element*** from the left on another dequeue operation.

Front and Display

- Get the front element from the queue i.e. arr[front] if the queue is not empty.
- Print all elements of the queue. If the queue is non-empty, traverse and print all the elements from the index front to rear.



Priority Queues

Priority Queue

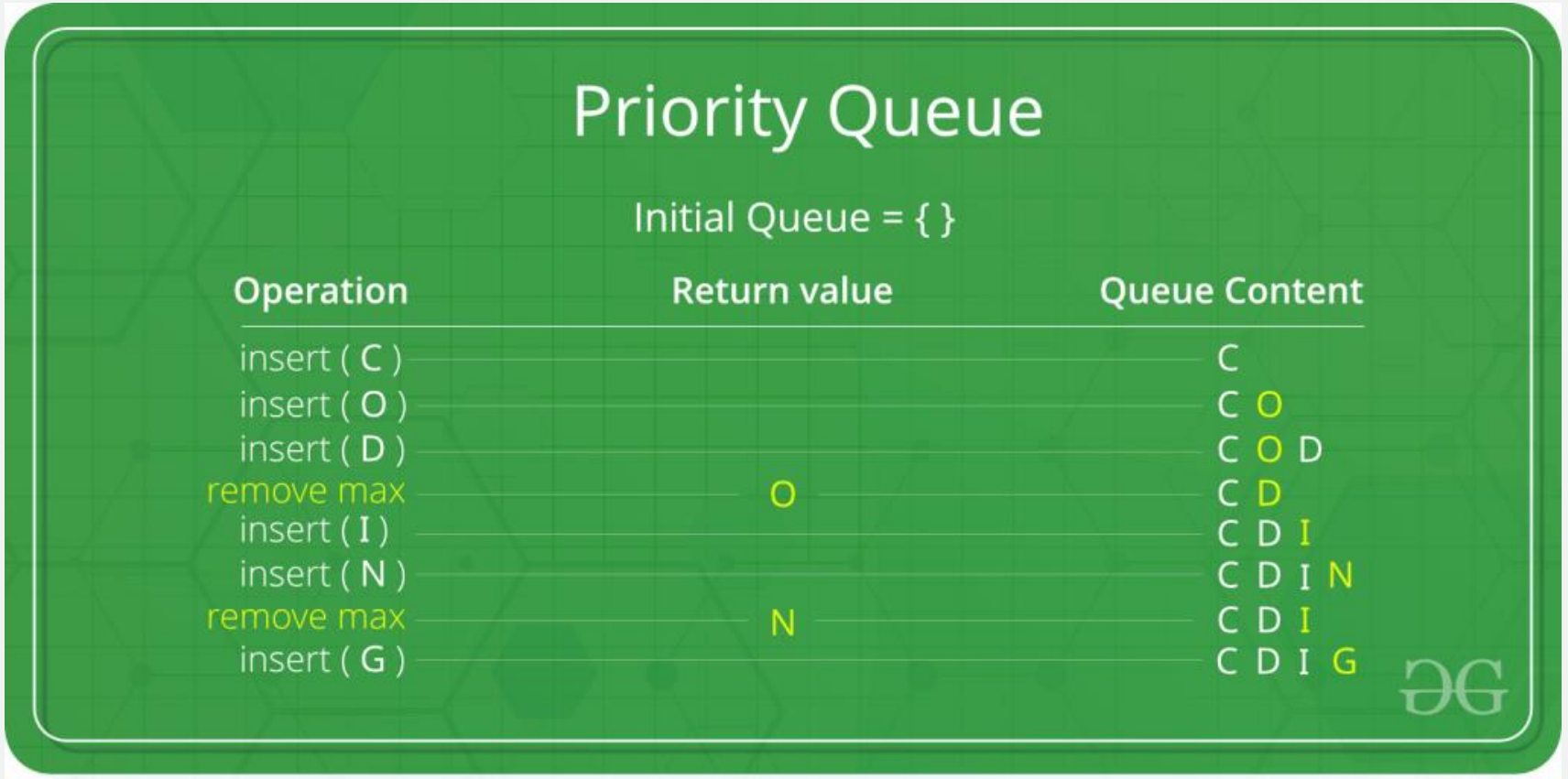
- Priority Queues are abstract data structures where each ***data/value in the queue has a certain priority.***
- For example, In airlines, baggage with the title “Business” or “First-class” arrives earlier than the rest.
- Various applications of the Priority queue in Computer Science are:
  - *Job Scheduling algorithms, CPU and Disk Scheduling, managing resources that are shared among different processes, etc.*

Priority Queue

- A priority Queue is an ***extension of the queue*** with the following properties:
  - Every item has a priority associated with it.
  - An element with high priority is dequeued before an element with low priority.
  - If two elements have the same priority, they are served according to their order in the queue.

Priority Queue

In the priority queue, an element with a maximum ASCII value will have the highest priority. The elements with higher priority are served first.



Insertion Operations of a Priority Queue

- **Create a new element:** Create a new element to be inserted into the queue. This element should include the data value to be stored in the queue and its associated priority value.
- **Add the element to the queue:** Append the new element to the end of the queue. This temporarily violates the heap property of the queue.
- **Restore the heap property:** Perform heapify operations to restore the heap property of the queue. This involves recursively comparing the newly added element with its parent nodes and swapping them if their priorities are not in order.

How is Priority assigned?

- In a priority queue, generally, the value of an element is considered for assigning the priority.
- For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority.
- The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

Deletion Operations of a Priority Queue

- Deleting an element from a priority queue involves *removing the element with the highest priority while maintaining the heap property of the queue.*
- **Identify the highest priority element:** Locate the element with the highest priority in the queue. This is typically the root node in a max heap or the last element in a min heap.
- **Remove the highest priority element:** Extract the identified element from the queue. This creates an empty slot in the heap structure.

Deletion Operations of a Priority Queue

- **Fill the empty slot:** To maintain the heap property, a new element needs to be placed in the empty slot. This is done by moving the last element of the heap to the empty slot and then performing heapify operations to ensure that the heap property is restored.
- **Heapify:** The heapify operation involves recursively comparing the new element with its parent nodes and swapping them if necessary until the heap property is restored. This ensures that the heap remains sorted after the deletion.

Operations of a Priority Queue

- **Peek in a Priority Queue**
  - This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

Types of Priority Queue

- **Ascending Order Priority Queue:** As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list.
- For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue.

Types of Priority Queue

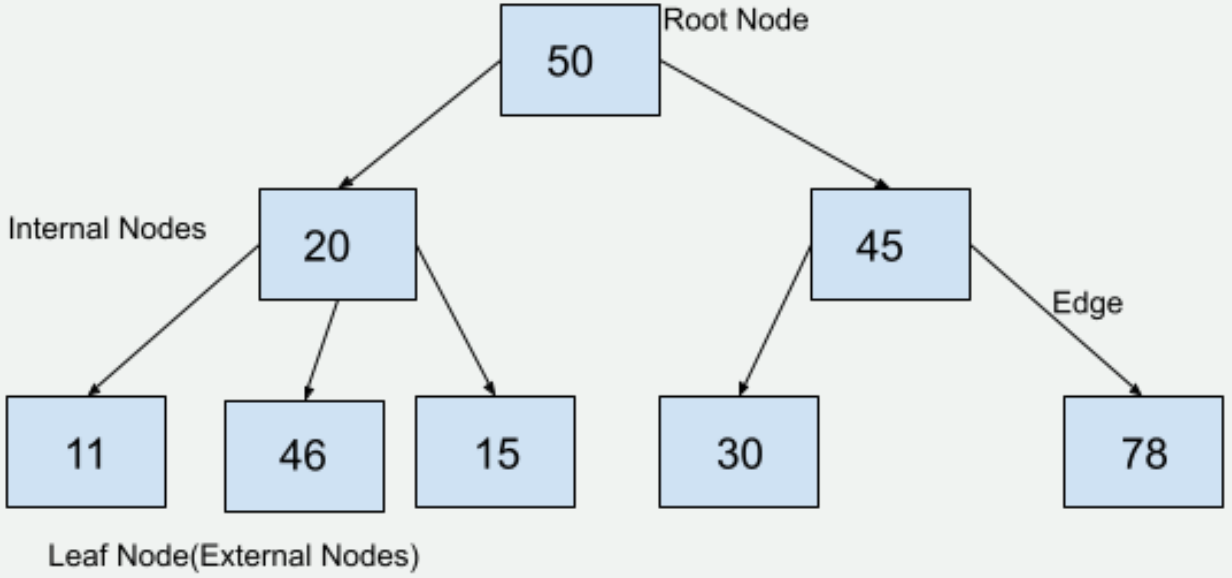
- **Descending order Priority Queue :** The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first.
- As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future.
- The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.

Difference between Priority Queue and Normal Queue?

- In Queue, the oldest element is dequeued first. While, in Priority Queue, an element based on the highest priority is dequeued.
- When elements are popped out of a priority queue the result obtained is either sorted in Increasing order or in Decreasing Order. While, when elements are popped from a simple queue, a FIFO order of data is obtained in the result.

Tree Data Structure

- A Tree is a Data structure in which data items are connected using references in a hierarchical manner.
- Each Tree consists of a root node from which we can access each element of the tree.
- Starting from the root node, each node contains zero or more nodes connected to it as children.
- A simple tree can be depicted as seen in the following figure.



Parts of a Tree Data structure

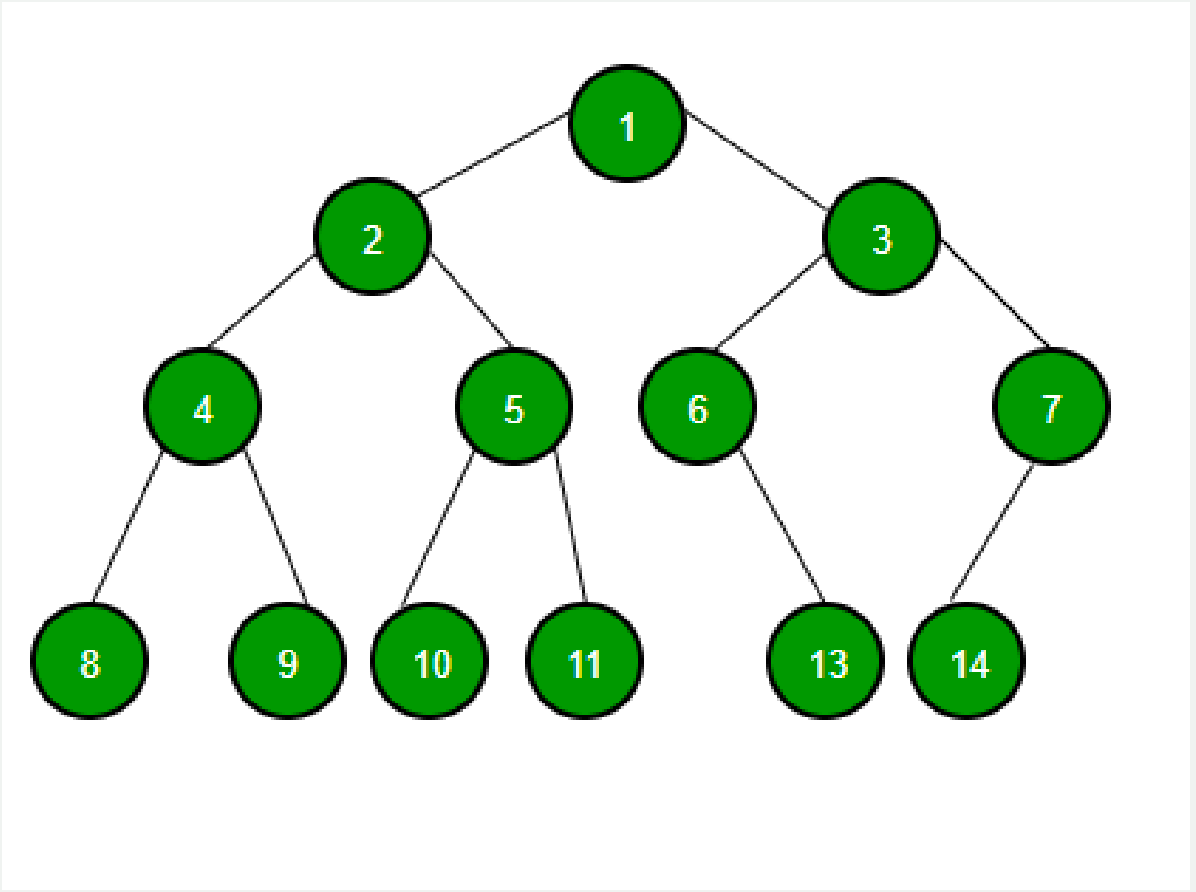
- **Root Node**: Root node is the topmost node of a tree. It is always the first node created while creating the tree and we can access each element of the tree starting from the root node. In the above example, the node containing element 50 is the root node.
- **Parent Node:** The parent of any node is the node which references the current node. In the above example, 50 is the parent of 20 and 45, 20 is parent of 11, 46 and 15. Similarly 45 is the parent of 30 and 78.
- **Child Node:** Child nodes of a parent node are the nodes at which the parent node is pointing using the references. In the example above, 20 and 45 are children of 50. The nodes 11, 46, and 15 are children of 20 and 30 and 78 are children of 45.

Parts of a Tree Data structure

- **Edge:** The reference through which a parent node is connected to a child node is called an edge. In the above example, each arrow that connects any two nodes is an edge.
- **Leaf Node:** These are those nodes in the tree which have no children. In the above example, 11, 46, 15, 30, and 78 are leaf nodes.
- **Internal Nodes:** Internal Nodes are the nodes which have at least one child. In the above example, 50, 20 and 45 are internal nodes.

Binary Trees

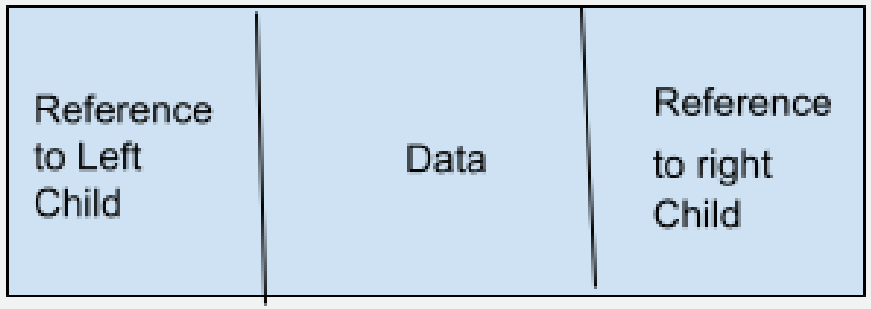
- Binary Tree is defined as **a tree data structure** with at most 2 children. Since each element in a binary tree can have **only 2 children**, we typically name them the left and right child.



Binary Tree Representation

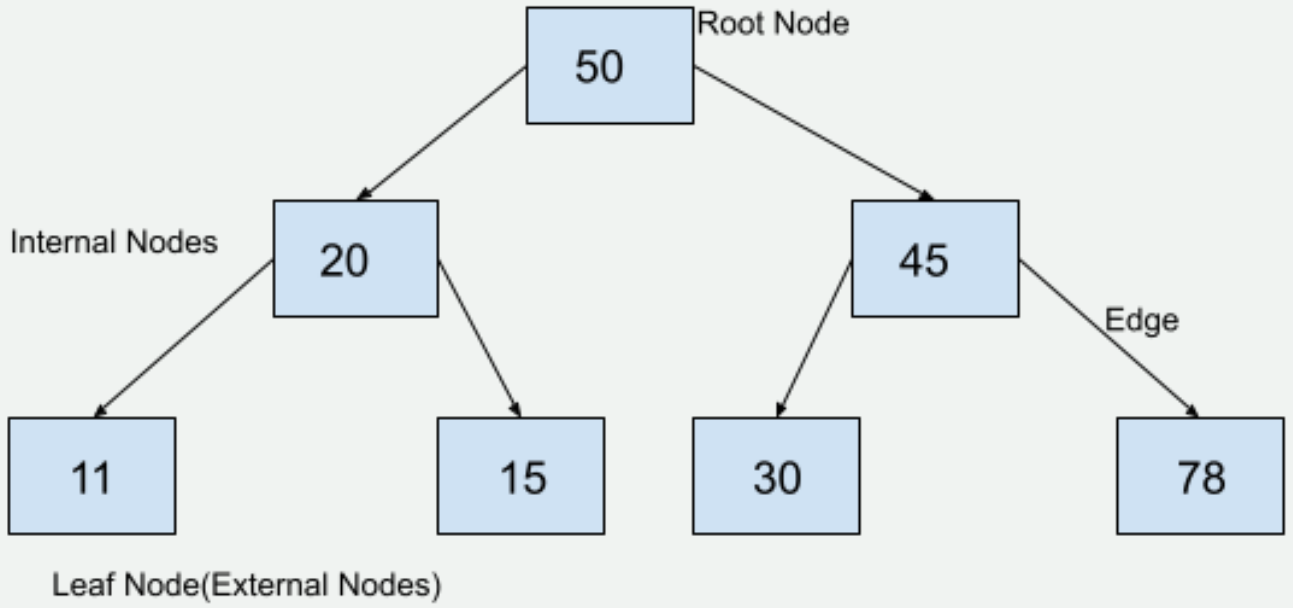
- A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.
- Binary Tree node contains the following parts:
  - **Data**
  - **Pointer to left child**
  - **Pointer to right child**

Binary Tree



- A binary tree is a tree data structure in which each node can have a maximum of 2 children.
- It means that each node in a binary tree can have either one, or two or no children.
- Each node in a binary tree contains data and references to its children. Both the children are named as left child and the right child according to their position.

Example


50
Root Node
Internal Nodes
20
45
Edge
11
15
30
78
Leaf Node(External Nodes)

Basic Operation On Binary Tree

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing an element.
- **Auxiliary Operation On Binary Tree:**
  - Finding the height of the tree
  - Find the level of the tree
  - Finding the size of the entire tree.

Arrays

- The array is a collection of the same type of elements at ***contiguous memory locations under the same name.***
- It is easier to access the element in the case of an array.
- The ***size*** is the key issue in the case of an array which must be known in advance so as to store the elements in it.
- ***No modification*** is possible at the runtime after the array is created and memory wastage can also occur if the size of the array is greater than the number of elements stored in the array

Dictionaries

- A dictionary is a collection of data values.
- It holds a ***key: value pair*** in which we can easily access a value if the key is known.
- It improves the readability of your code and makes it easier to debug.
- It is fast as the access of a value through a key is a constant time operation.
- A dictionary is also called a hash, a map, a ***HashMap*** in different programming languages.
- Keys in a dictionary must be unique an attempt to create a duplicate key will typically overwrite the existing value for that key.

Dictionaries

- Dictionary is an abstract data structure that supports the following operations:
  - ***Search (K key) (returns the value associated with the given key)***
  - ***Insert (K key, V value)***
  - ***Delete (K key)***
- Each element stored in a dictionary is identified by a key of type K.
- Dictionary represents a mapping from keys to values.
- Other terms for keyed containers include the names *map, table, search table, associative array, or hash.*

Comparison Between Array and Dictionary:

S.N.	Array	Dictionary
1.	Stores just a set of objects.	Represents the relationship between pair of objects
2.	Lookup time is more in the case of array O(N), where N is the size of the array.	Lookup time is less compared to an array. Generally, it is O(1)
3.	Elements are stored at contagious memory locations.	Elements may or may not be stored at a contagious memory location.
4.	Items are unordered, changeable, and do allow duplicates.	Items are ordered, changeable, and do not allow duplicates.

Comparison Between Array and Dictionary:

S.N.	Array	Dictionary
5.	Items are not represented as key: value pair	Items are represented as key: value pair
6.	The values in the array are of the same data type	The values in dictionary items can be of any data type
7.	Values can be accessed randomly without the need for any key	To access a value the key is required.

Sets and disjoint sets union

- **Set:** A set is a collection of distinct elements. The Set can be represented, for examples, as S1= {1,2,5,10}.
- **Disjoint Sets:** The disjoints sets are those do not have any common element. For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.
- Two sets are called disjoint sets if they don't have any element in common, the intersection of sets is a null set.
- **Disjoint Set Operations:** The disjoint set operations are:
  - **Union**
  - **Find**

Sets and disjoint sets union

- **Disjoint set Union:** If S_i and S_j are tow disjoint sets, then their union $\mathbf{S_i \cup S_j}$ consists of all the elements x such that x is in S_i or S_j.
- Example: $S_1 = \{1,7,8,9\}$ and $S_2 = \{2,5,10\}$, so $S_1 \cup S_2 = \{1,2,5,7,8,9,10\}$
- Find: Given the element I, find the set containing i.
- **Example:**
- S1={1,7,8,9} S2={2,5,10} S3={3,4,6}
- Find(4)= S_3 Find(5)=S_2 Find(7)=S_1

Q1. Briefly Explain the Queue data structure. Write an algorithm to add and remove an element from the circular queue and compute the complexity of your algorithm.

Queue Data Structure

Queue Data Structure

- *A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.*
- We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.
- The element which is first pushed into the order, the operation is first performed on that.

Write an algorithm to add and remove an element from the circular queue and compute the complexity of your algorithm.

- A *circular queue is similar to a linear queue* as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.
- The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.
- Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

Write an algorithm to add and remove an element from the circular queue and compute the complexity of your algorithm.

- A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.
- The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.
- Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

Algorithm to insert an element in a circular queue

The steps of enqueue operation are given below:

First, we will check whether the Queue is full or not.

Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.

When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Algorithm to insert an element in a circular queue

- **Step 1:** IF (REAR+1)%MAX = FRONT
  Write " OVERFLOW "
  Goto step 4
  [End OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE IF REAR = MAX - 1 and FRONT ! = 0
  SET REAR = 0
  ELSE
  SET REAR = (REAR + 1) % MAX
  [END OF IF]
- **Step 3:** SET QUEUE[REAR] = VAL
- **Step 4:** EXIT

Algorithm to remove an element in a circular queue

The steps of dequeue operation are given below:

First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.

When the element is deleted, the value of front gets decremented by 1.

If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to remove an element in a circular queue

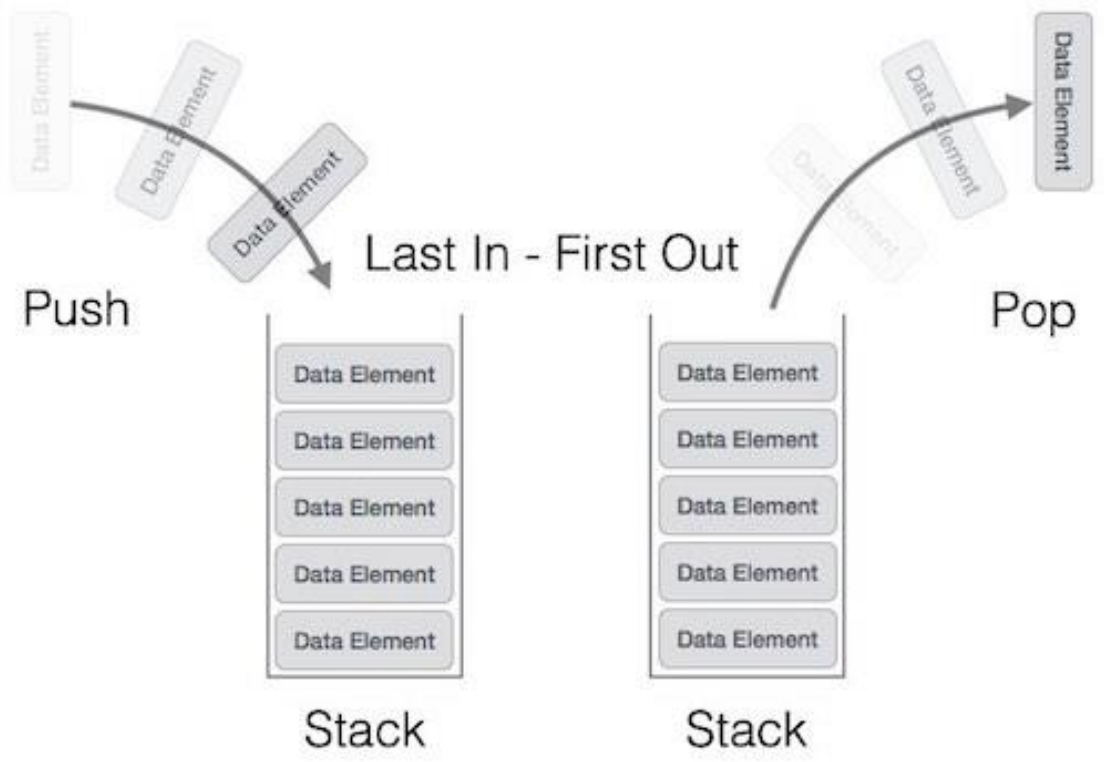
```
Step 1: IF FRONT = -1
        Write " UNDERFLOW "
        Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
        SET FRONT = REAR = -1
ELSE
        IF FRONT = MAX -1
                SET FRONT = 0
        ELSE
                SET FRONT = FRONT + 1
        [END of IF]
[END OF IF]
Step 4: EXIT
```

Q1. Briefly Explain the Stack data structure. Write an algorithm to add and remove an element from the stack and compute the complexity of your algorithm.

Stack Data Structure

- 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).
- Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations
- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

Stack Data Structure



Stack Data Structure

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Stack Data Structure

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```