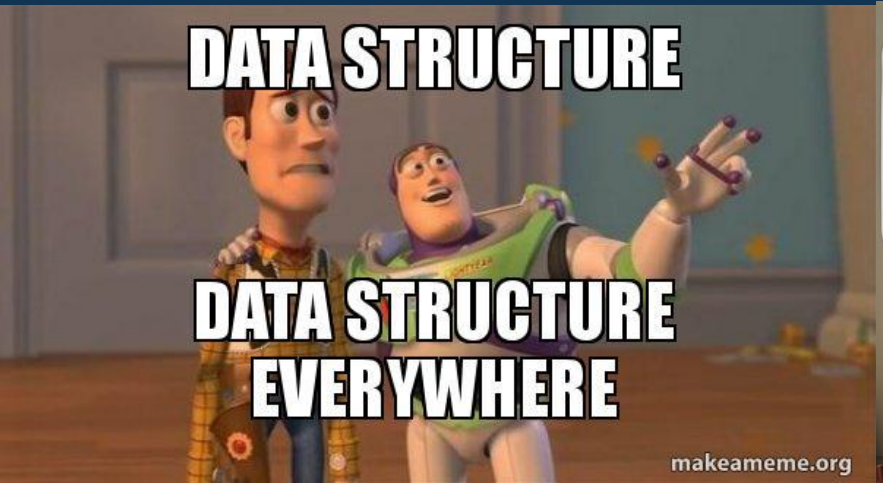
HyperionDev

Linear Data Structures: Stacks and Queues

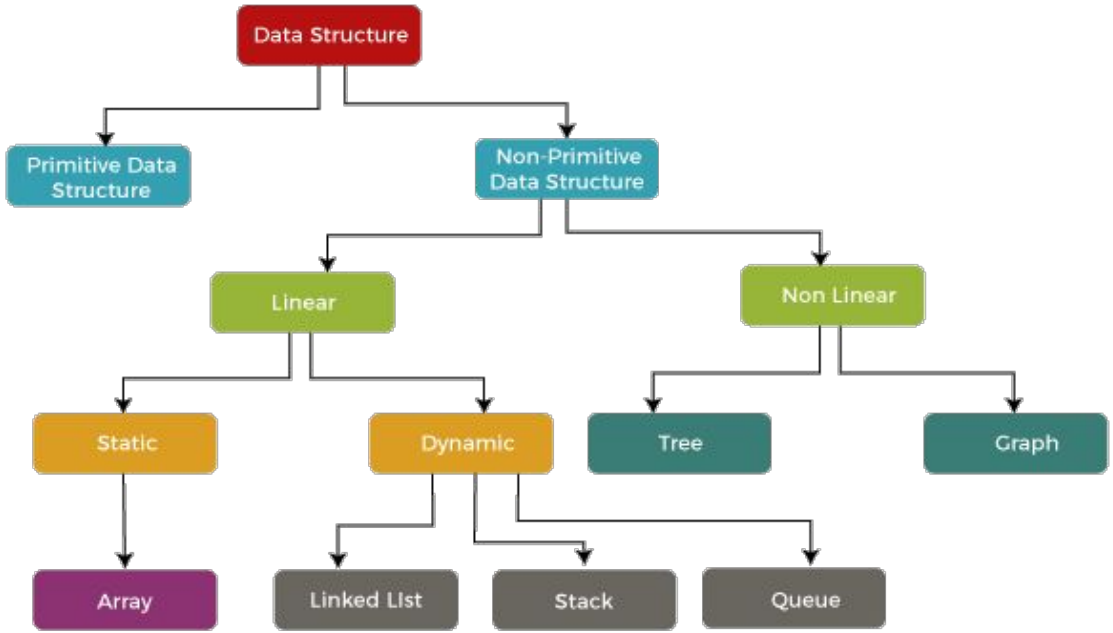
- Abstract Data Types vs Data Structures
- Stacks
- Queues
- Priority Queues

What is a data structure?

A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.

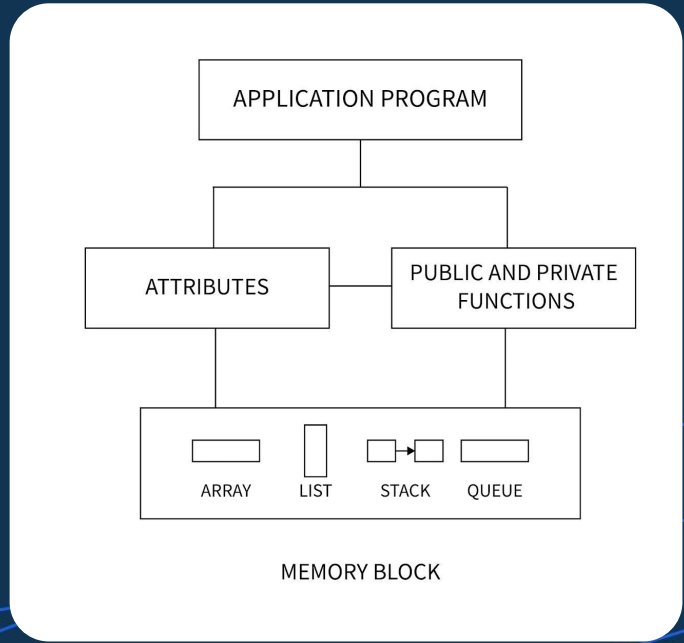


Types of data structures



Abstract data types (ADTs)

An **Abstract Data Type** in data structure is a data type whose behavior is defined with the help of some attributes and some functions.

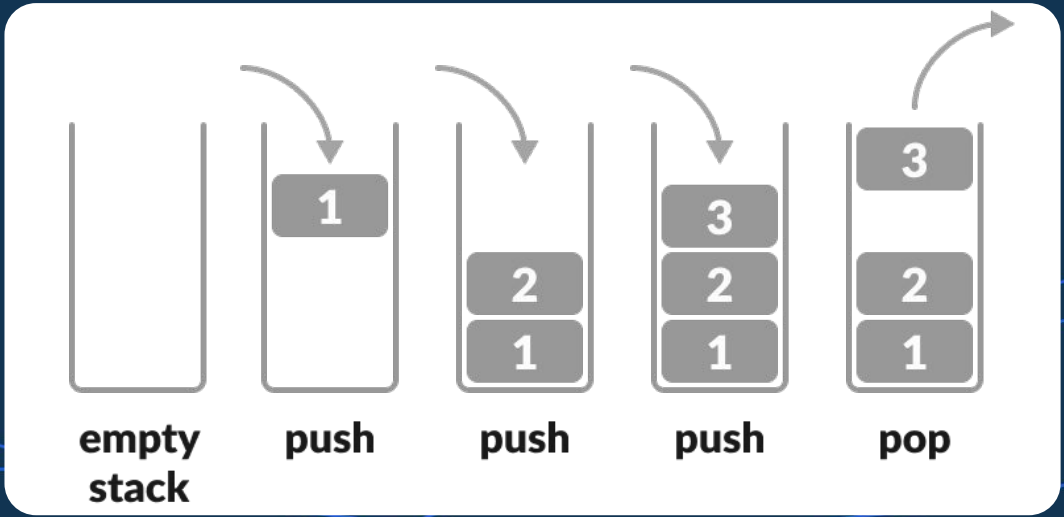


Abstract data types vs Data structures

Abstract data type	Data structure
• ADT is a logical description of a new type. • Defined as a set of data values with operations. • It does not provide the implementation of operations.	• Data structures are implementations of ADT. • They are a concrete data type whereas ADT are just representations.

Stack

Stack is a linear data structure that 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).



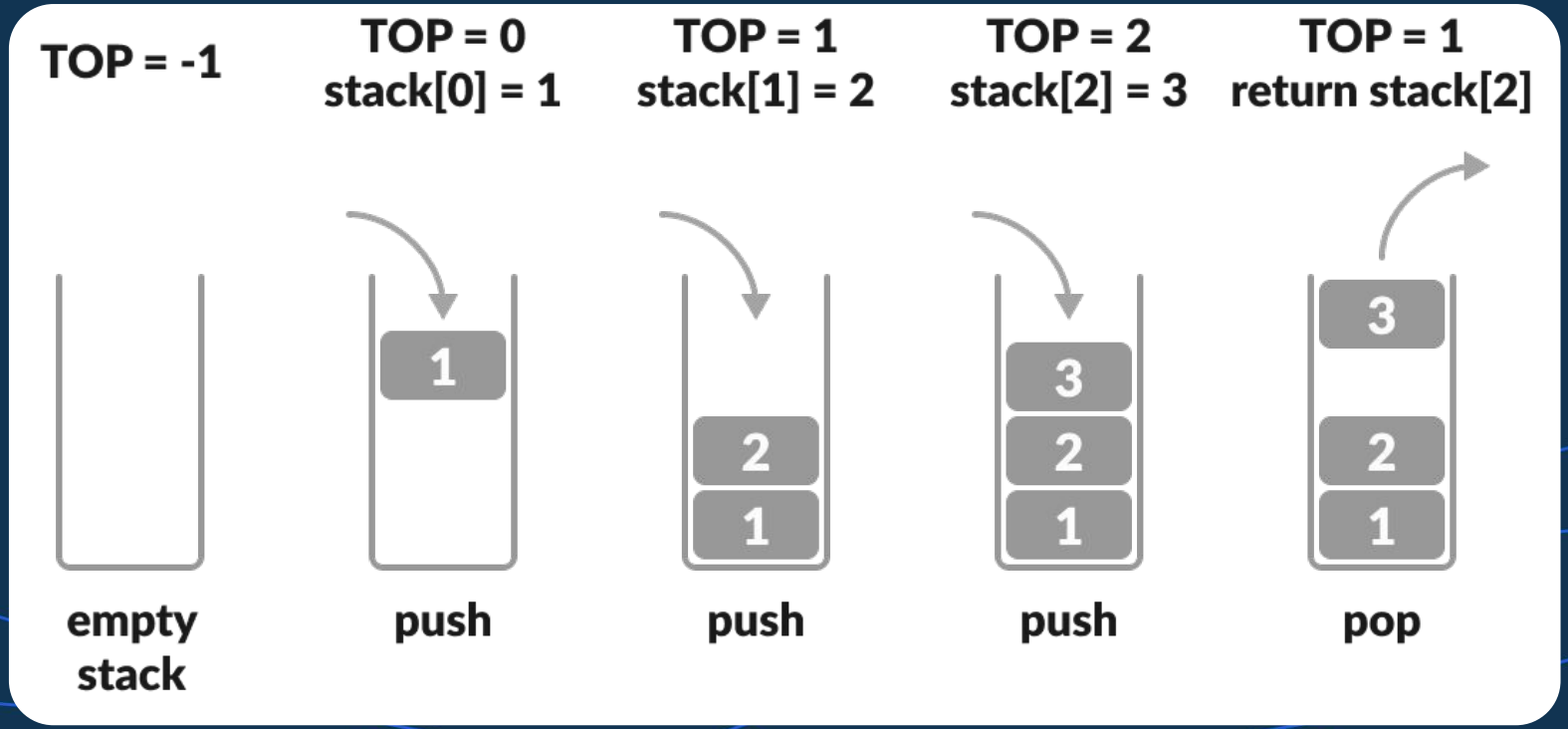
Advantages of Stacks

- Efficient data management
- Efficient management of functions
- Control over memory
- Smart memory management
- Not easily corrupted
- Does not allow resizing of variables

Disadvantages of Stacks

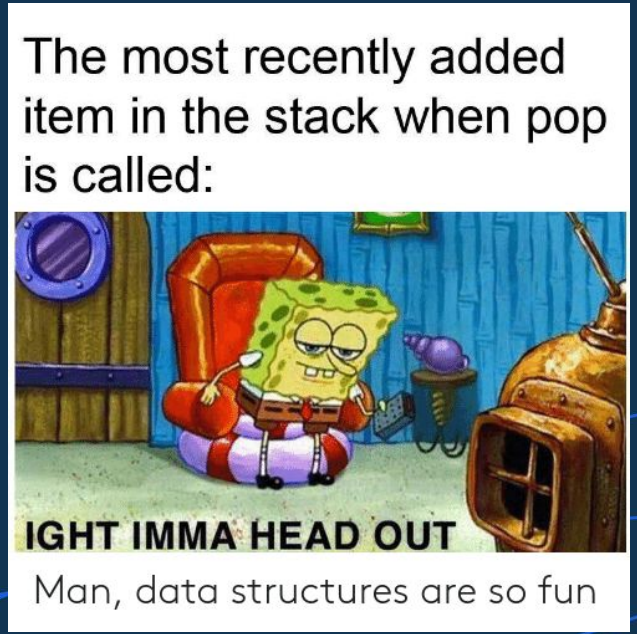
- Limited memory size
- Chances of stack overflow
- Random access is not possible
- Unreliable
- Undesired termination

Working with Stacks



Basic Operations on Stacks

- IsEmpty: Check if the stack is empty
- IsFull: Check if the stack is full
- Push: Add an element to the top of a stack
- Pop: Remove an element from the top of a stack
- Peek: Get the value of the top element without removing it



Stack implementation using arrays

JavaScript	Python

class Stack {
constructor() {
this.MAX = 1000
this.items = new Array(MAX).fill(0);
this.top = -1;
}
	class Stack:
	def __init__(self):
	self.MAX = 1000
	self.items = [0] * MAX
	self.top = -1

Stack implementation using arrays: isEmpty() & isFull()

JavaScript	Python

isEmpty() {
  return this.top < 0
}
 	
isFull() {
  return this.top >= (this.MAX - 1)
}
	def isEmpty(self):
	  return self.top < 0
	
	def isFull(self):
	  return self.top >= (self.MAX - 1)

Stack implementation using arrays: push()

JavaScript	Python

push(item) {
  if (this.isFull()) {
  console.log(“Stack Overflow”);
  return false;
  } else {
  this.top += 1;
  this.items[this.top] = item;
  console.log(`${item} pushed into stack`)
  return true
  }
}
	def push(self, item):
	  if(self.isFull()):
	  print(“Stack Overflow”)
	  return False;
	  else:
	  self.top += 1
	  self.items[self.top] = item
	  print(f”{item} pushed into stack”)
	  return True

Stack implementation using arrays: pop()

JavaScript	Python

pop() {
  if (this.isEmpty()) {
  console.log(“Stack Underflow”);
  return false;
  } else {
  let item = this.items[this.top]
  this.top -= 1;
  return item
  }
}
	def pop(self):
	  if(self.isEmpty()):
	  print(“Stack Underflow”)
	  return False;
	  else:
	  item = self.items[self.top]
	  self.top -= 1
	  return item

Stack implementation using arrays: peek()

JavaScript	Python

peek() {
  if (this.isEmpty()) {
  console.log(“Stack Underflow”);
  return false;
  } else {
  let item = this.items[this.top]
  return item
  }
}
	def peek(self):
	  if(self.isEmpty()):
	  print(“Stack Underflow”)
	  return False;
	  else:
	  item = self.items[self.top]
	  return item

Advantages of array implementation of Stacks

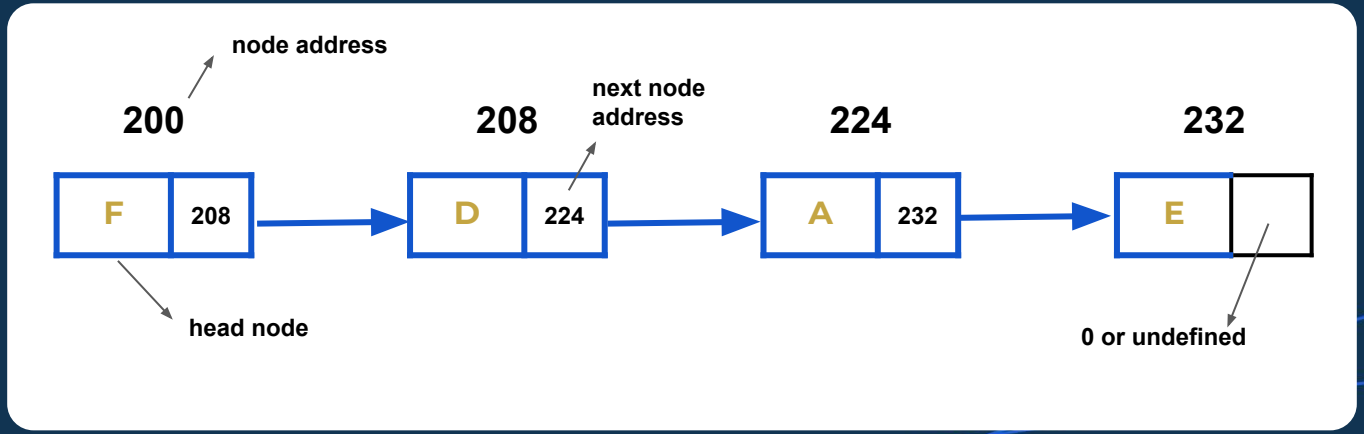
- Easy to implement
- Memory is saved as pointers are not involved

Disadvantages of array implementation of Stacks

- It is not dynamic
- Size need to be defined beforehand

Linked lists

A **linked list** is a linear data structure, in which the elements are not stored at contiguous memory locations.



Stack implementation using linked list

JavaScript	Python

class Node {
  constructor(data) {
  this.data = data;
  this.next = null;
  }
}
class Stack {
  constructor() {
  this.top = null;
  }
}
	class Node:
	  def __init__(self, data):
	  self.data = data
	  self.next = None
	
	class Stack:
	  def __init__(self):
	  self.top = null

Stack implementation using arrays: isEmpty()

JavaScript	Python

isEmpty() {
  if (this.top == null){
  return true;
  } else {
  return false
  }
}
	def isEmpty(self):
	  if self.top == None:
	  return True
	  else:
	  return False

Stack implementation using linked list: push()

JavaScript	Python

push(data){
  if (this.top == null){
  this.top = new Node(data)
  } else {
  let node = new Node(data)
  node.next = this.top
  this.top = node
  }
}
	def push(self, data):
	  if self.top == None:
	  self.top = Node(data)
	  else:
	  node = Node(data)
	  node.next = self.top
	  self.top = node

Stack implementation using linked list: pop()

JavaScript	Python

pop(){
  if (this.isEmpty()){
  return;
  } else {
  let node = this.top;
  this.top = this.top.next;
  node.next = null
  return node.data
  }
}
	def pop(self):
	  if self.isEmpty():
	  return None
	  else:
	  node = self.top
	  self.top = self.top.next
	  node.next = None
	  return node.data

Stack implementation using linked list: peek()

JavaScript	Python

peek(){
if (this.isEmpty()){
return;
} else {
return this.top.data
}
}
	def peek(self):
	if self.isEmpty():
	return None
	else:
	return self.top.data

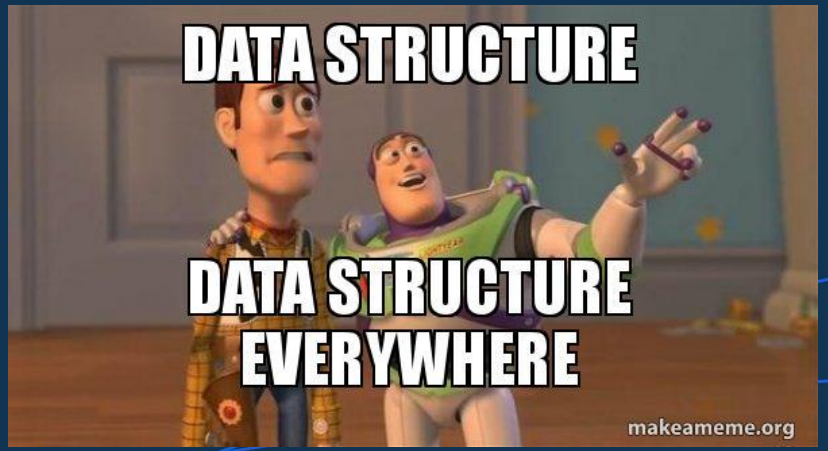
Time complexities of stack operations

Operations	Complexity
isEmpty	O(1)
push()	O(1)
pop()	O(1)
peek()	O(1)

Real world use cases of Stacks

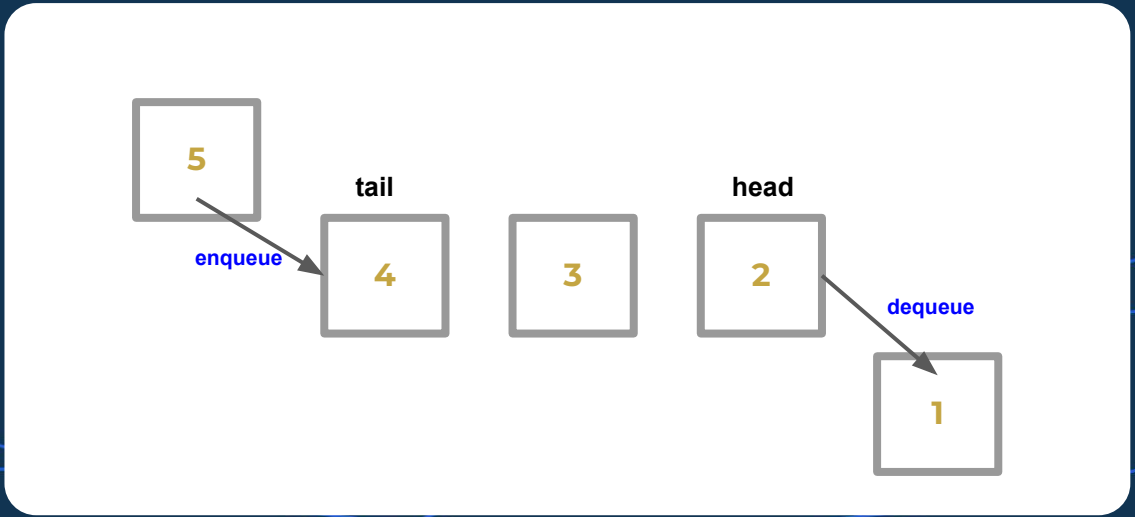
Stacks are used:

- To reverse a word
- In compilers
- In browsers



Queues

Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).



Advantages of Queues

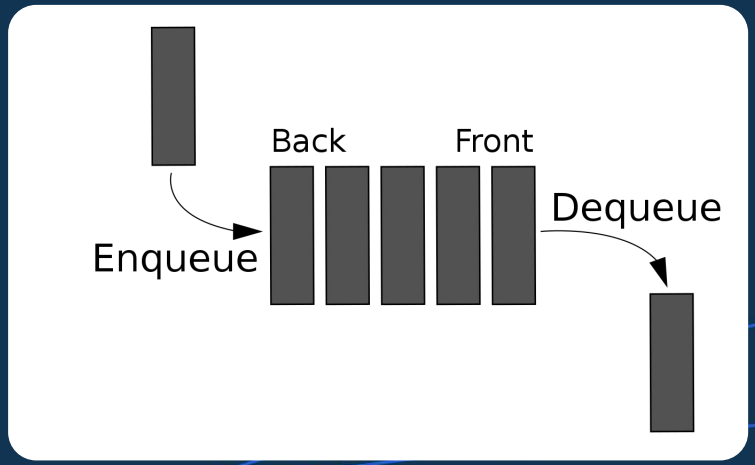
- Efficient data management
- Operations can be performed with ease
- Useful when used by multiple consumers
- Speed
- Can be used to implement other data structures
- Does not allow resizing of variables

Disadvantages of Queues

- Inserting and deleting of elements in the middle is time consuming
- Limited space
- Searching an element takes O(n) time
- Size must be defined beforehand

Basic Operations on Queues

- isEmpty(): Determines whether the queue is empty or not
- isFull(): Determines whether the queue is full or not.
- enqueue(): Adds an item to the queue.
- dequeue(): Removes an item from the queue.
- getFront(): Get the front item from queue.
- getRear(): Get the last item from queue.



Queue implementation using arrays

JavaScript	Python

class Queue {
constructor(capacity) {
this.capacity = capacity;
this.size = 0;
this.front = 0;
this.rear = capacity - 1
this.items = new Array(capacity);
}
	class Queue:
	def __init__(self, capacity):
	self.capacity = capacity
	self.size = 0
	self.front = 0
	self.rear = capacity - 1
	self.items = [None] * capacity

Queue implementation using arrays: isEmpty() & isFull()

JavaScript	Python

isEmpty() {
  return this.size == 0;
}
isFull() {
  return this.size == this.capacity;
}
	def isEmpty(self):
	  return self.size == 0
	
	def isFull(self):
	  return self.size == self.capacity

Queue implementation using arrays: enqueue()

JavaScript	Python

enqueue(item) {
  if (this.isFull()) {
  console.log(“Full”);
  return;
  }
  this.rear = (this.rear + 1) % this.capacity
  this.items[this.rear] = item
  this.size += 1
}
	def enqueue(self, item):
	  if self.isFull():
	  print(“Full”)
	  return
	  self.rear = (self.rear + 1) % self.capacity
	  self.items[self.rear] = item
	  self.size += 1

Queue implementation using arrays: dequeue()

JavaScript	Python

dequeue(item) {
  if (this.isEmpty()) {
  console.log(“Empty”);
  return;
  }
  this.front = (this.front+ 1) % this.capacity
  this.size -= 1
}
	def dequeue(self, item):
	  if self.isEmpty():
	  print(“Empty”)
	  return
	  self.front = (self.front + 1) % self.capacity
	  self.size -= 1

Queue implementation using arrays: getFront() & getRear()

JavaScript	Python

getFront() {
  if (this.isEmpty()) {
  console.log(“Empty”);
  }
  return this.items[this.front]
}
getRear() {
  if (this.isEmpty()) {
  console.log(“Empty”);
  }
  return this.items[this.rear]
}
	def getFront(self):
	  if self.isEmpty():
	  print(“Empty”)
	  return self.items[self.front]
	
	def getRear(self):
	  if self.isEmpty():
	  print(“Empty”)
	  return self.items[self.rear]

Advantages of array implementation of queues

- Easy to implement
- Operations are performed with ease because of FIFO rule.
- Efficient data management

Disadvantages of array implementation of queues

- Fixed size
- Size must be defined beforehand

Queue implementation using linked list

JavaScript	Python

class Node {
constructor(data) {
this.data = data;
this.next = null;
}
}
class Queue {
constructor() {
this.front = null;
this.rear = null;
This.size = 0
}
}
	class Node:
	def __init__(self, data):
	self.data = data
	self.next = None
	
	class Queue:
	def __init__(self):
	self.front = None
	self.rear = None
	self.size = 0

Queue implementation using linked list: isEmpty()

JavaScript	Python

isEmpty() {
  if (this.size == 0){
  return true;
  } else {
  return false;
  }
}
	def isEmpty(self):
	  if self.size == None:
	  return True
	  else:
	  return False

Queue implementation using linked list: enqueue()

JavaScript	Python

enqueue(item) {
  let node = new Node(item)
  if (this.isEmpty()) {
  this.front = node;
  this.rear = node
  } else {
  this.rear.next = node;
  this.rear = node
  }
  this.size++;
}
	def enqueue(self, item):
	  node = Node(item)
	  if self.isEmpty():
	  self.front = node
	  self.rear = node
	  else:
	  self.rear.next = node
	  self.rear = node
	  self.size += 1

Queue implementation using linked list: dequeue()

JavaScript	Python

dequeue() {
  if (this.isEmpty()) {
  return null
  }
  let itemToBeRemoved = this.front
  if (this.front == this.rear){
  this.rear = null
  }
  this.front = this.front.next
  this.size--
}
	def dequeue(self):
	  if self.isEmpty():
	  return None
	  itemToBeRemoved = self.front
	  if self.front == self.rear:
	  self.rear = None
	  self.front = self.front.next
	  self.size -= 1

Queue implementation using linked list: getFront() & getRear()

JavaScript	Python

getFront() {
  if (this.isEmpty()) {
  console.log(“Empty”);
  }
  return this.front
}
getRear() {
  if (this.isEmpty()) {
  console.log(“Empty”);
  }
  return this.rear
}
	def getFront(self):
	  if self.isEmpty():
	  print(“Empty”)
	  return self.front
	
	def getRear(self):
	  if self.isEmpty():
	  print(“Empty”)
	  return self.rear

Time complexities of queue operations

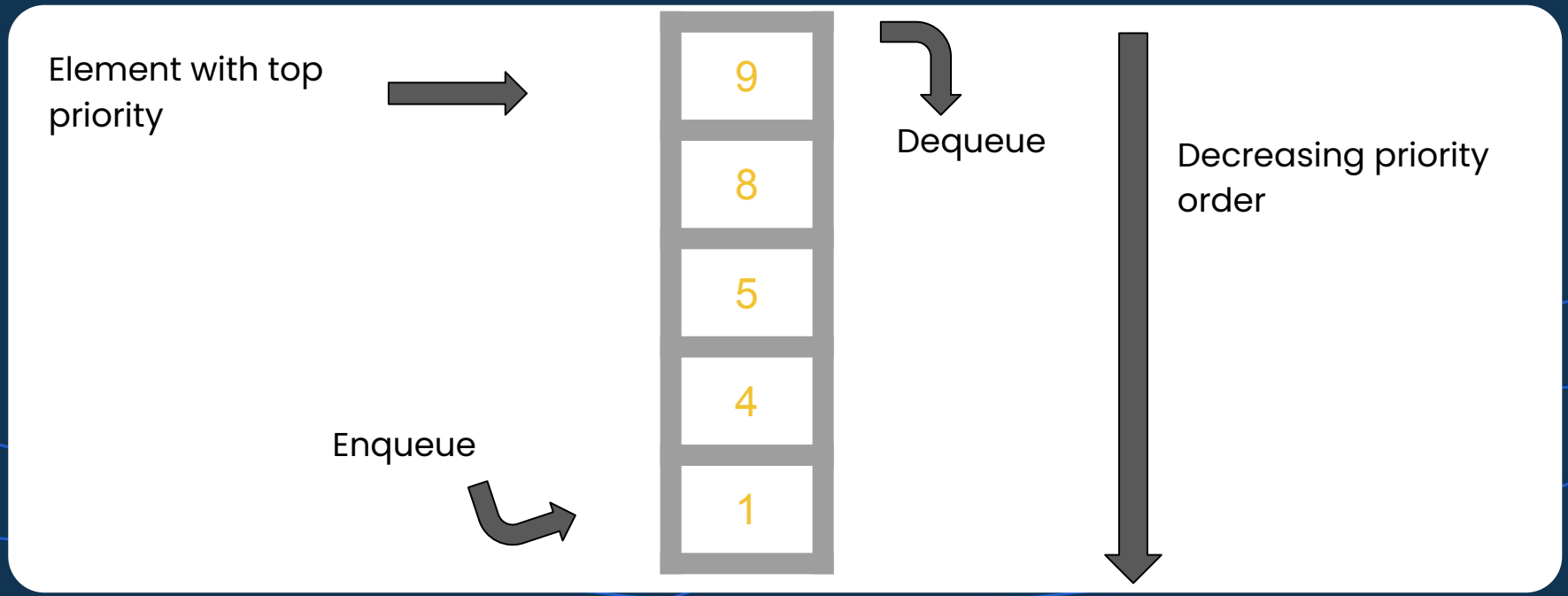
Operations	Complexity
isEmpty()	O(1)
isFull()	O(1)
enqueue()	O(1)
dequeue()	O(1)
getFront()	O(1)
getRear()	O(1)

Array-based queue vs Linked list-based queue

Array-based queue	Linked list-based queue
Faster compared to list-based queue	It is slower as compared to array-based queues.
Elements can be accessed randomly	Elements can be accessed sequentially only.
Requires less memory	It requires more memory.
The size of the queue should be known in advance.	It’s not necessary to know the size of the queue in advance.
Resizing array-based queues is complex.	Resizing is simple.
Insertion at the beginning is difficult.	Insertion at both end and beginning is easy.

Priority Queues

A **priority queue** is a special type of queue in which each element is associated with a priority and is served according to its priority.



Real world use cases of Queues

Queues are used:

- In CPU scheduling
- In routers and switches
- In networking
- in maintaining the playlist in media players
- To handle interrupts in the operating system

Q & A

THANK YOU

HyperionDev