## Topics

1. Create Queue Interface

public interface Queue<E> {

boolean isEmpty();

int size();

E first();

void enqueue(E element);

E dequeue();

}

1. Create Queue Using Array

public class ArrayQueue<E> implements Queue<E> {

private static final int DEFAULT\_CAPACITY = 10;

private E[] queueArray;

private int front;

private int rear;

private int size;

public ArrayQueue() {

this(DEFAULT\_CAPACITY);

}

public ArrayQueue(int capacity) {

if (capacity <= 0) {

throw new IllegalArgumentException("Capacity must be positive");

}

queueArray = (E[]) new Object[capacity];

front = 0;

rear = -1;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return queueArray[front];

}

public void enqueue(E element) {

if (size == queueArray.length) {

resize(2 \* queueArray.length);

}

rear = (rear + 1) % queueArray.length;

queueArray[rear] = element;

size++;

}

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E element = queueArray[front];

queueArray[front] = null;

front = (front + 1) % queueArray.length;

size--;

if (size > 0 && size == queueArray.length / 4) {

resize(queueArray.length / 2);

}

return element;

}

private void resize(int capacity) {

E[] newArray = (E[]) new Object[capacity];

for (int i = 0; i < size; i++) {

newArray[i] = queueArray[(front + i) % queueArray.length];

}

queueArray = newArray;

front = 0;

rear = size - 1;

}

}

1. Create Queue Using Linked Lists

public class LinkedQueue<E> implements Queue<E> {

private Node<E> front;

private Node<E> rear;

private int size;

private static class Node<E> {

private E element;

private Node<E> next;

public Node(E element, Node<E> next) {

this.element = element;

this.next = next;

}

}

public LinkedQueue() {

front = null;

rear = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public E first() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

return front.element;

}

public void enqueue(E element) {

Node<E> newNode = new Node<>(element, null);

if (isEmpty()) {

front = newNode;

} else {

rear.next = newNode;

}

rear = newNode;

size++;

}

public E dequeue() {

if (isEmpty()) {

throw new IllegalStateException("Queue is empty");

}

E element = front.element;

front = front.next;

size--;

if (isEmpty()) {

rear = null;

}

return element;

}

}

1. Implement Basic Methods of Queue

* isEmpty()
* size()
* first()
* enqueue(E e)
* dequeue()

public class Main {

public static void main(String[] args) {

Queue<Integer> arrayQueue = new ArrayQueue<>();

arrayQueue.enqueue(1);

arrayQueue.enqueue(2);

arrayQueue.enqueue(3);

System.out.println(arrayQueue.size());

System.out.println(arrayQueue.first());

arrayQueue.dequeue();

System.out.println(arrayQueue.size());

System.out.println(arrayQueue.first());

Queue<String> linkedQueue = new LinkedQueue<>();

linkedQueue.enqueue("Hello");

linkedQueue.enqueue("World");

System.out.println(linkedQueue.size());

System.out.println(linkedQueue.first());

linkedQueue.dequeue();

System.out.println(linkedQueue.size());

System.out.println(linkedQueue.first());

}

}

## Homework

1. Augment the ArrayQueue implementation with a new rotate( ) method having semantics identical to the combination, enqueue(dequeue( )). But, your implementation should be more efficient than making two separate calls (for example, because there is no need to modify the size).

class ArrayQueue:

def \_\_init\_\_(self):

self.\_data = []

self.\_front = 0

def is\_empty(self):

return len(self.\_data) == 0

def enqueue(self, item):

self.\_data.append(item)

def dequeue(self):

if self.is\_empty():

raise IndexError("Queue is empty")

item = self.\_data[self.\_front]

self.\_front += 1

return item

def rotate(self):

if self.is\_empty():

return

self.\_data = self.\_data[self.\_front:] + self.\_data[:self.\_front]

self.\_front = 0

1. Implement the clone( ) method for the ArrayQueue class.

class ArrayQueue:

def \_\_init\_\_(self):

self.\_data = []

self.\_front = 0

def is\_empty(self):

return len(self.\_data) == 0

def enqueue(self, item):

self.\_data.append(item)

def dequeue(self):

if self.is\_empty():

raise IndexError("Queue is empty")

item = self.\_data[self.\_front]

self.\_front += 1

return item

def clone(self):

new\_queue = ArrayQueue()

new\_queue.\_data = self.\_data.copy()

new\_queue.\_front = self.\_front

return new\_queue

1. Implement a method with signature concatenate(LinkedQueue Q2) for the LinkedQueue class that takes all elements of Q2 and appends them to the end of the original queue. The operation should run in O(1) time and should result in Q2 being an empty queue.

class \_Node:

\_\_slots\_\_ = '\_element', '\_next'

def \_\_init\_\_(self, element, next\_node):

self.\_element = element

self.\_next = next\_node

class LinkedQueue:

def \_\_init\_\_(self):

self.\_front = None

self.\_tail = None

self.\_size = 0

def is\_empty(self):

return self.\_size == 0

def enqueue(self, item):

new\_node = \_Node(item, None)

if self.is\_empty():

self.\_front = new\_node

else:

self.\_tail.\_next = new\_node

self.\_tail = new\_node

self.\_size += 1

def dequeue(self):

if self.is\_empty():

raise IndexError("Queue is empty")

item = self.\_front.\_element

self.\_front = self.\_front.\_next

self.\_size -= 1

if self.is\_empty():

self.\_tail = None

return item

def concatenate(self, Q2):

if Q2.is\_empty():

return

if self.is\_empty():

self.\_front = Q2.\_front

else:

self.\_tail.\_next = Q2.\_front

self.\_tail = Q2.\_tail

self.\_size += Q2.\_size

Q2.\_front = None

Q2.\_tail = None

Q2.\_size = 0

1. Use a queue to solve the Josephus Problem.

from collections import deque

def josephus\_problem(n, k):

queue = deque(range(1, n + 1)) # Initialize the queue with numbers from 1 to n

while len(queue) > 1:

# Simulate the counting process by dequeuing and enqueuing individuals

for \_ in range(k - 1):

queue.append(queue.popleft())

# Eliminate the individual at the front of the queue

queue.popleft()

# The last remaining person in the queue is the survivor

return queue[0]

# Example usage

n = 7 # Number of people

k = 3 # Counting rule

survivor = josephus\_problem(n, k)

print("The survivor in the Josephus Problem with", n, "people and counting rule", k, "is:", survivor)

1. Use a queue to simulate Round Robin Scheduling.

from collections import deque

class Process:

def \_\_init\_\_(self, name, burst\_time):

self.name = name

self.burst\_time = burst\_time

def execute(self, quantum):

if self.burst\_time <= quantum:

print("Executing process", self.name, "for", self.burst\_time, "units")

self.burst\_time = 0

else:

print("Executing process", self.name, "for", quantum, "units")

self.burst\_time -= quantum

def is\_completed(self):

return self.burst\_time == 0

def round\_robin\_scheduling(processes, quantum):

queue = deque(processes)

while queue:

current\_process = queue.popleft()

current\_process.execute(quantum)

if not current\_process.is\_completed():

queue.append(current\_process)

# Example usage

processes = [

Process("P1", 10),

Process("P2", 5),

Process("P3", 8),

Process("P4", 2)

]

quantum = 3

round\_robin\_scheduling(processes, quantum)