**Pseudocode**

**Controller Main Methods**

**PROCEDURE createItem**(type: INTEGER, name: STRING, description: STRING, priority: INTEGER, day: INTEGER, month: INTEGER, year: INTEGER)

// Create a Gregorian calendar object using the provided date

calendar = NEW GregorianCalendar(year, month - 1, day)

// Create a new Item object with the given parameters

item = NEW Item(name, description, priority, calendar, type, priority)

// Add the new item to the itemHashTable

itemHashTable.addElement(name, item)

// Add the new item to priority queues based on priority and date

itemPriorityQueueByPriority.offer(item)

itemPriorityQueueByDate.offer(item)

// Create an action of type 0 (e.g., a creation action) for the new item

createAction(0, item)

END PROCEDURE

**PROCEDURE modifyItem**(newName: STRING, newDescription: STRING, newDay: INTEGER, newMonth: INTEGER, newYear: INTEGER, hashPointer: INTEGER, linkedListPointer: INTEGER)

// Create a new Gregorian calendar instance with the provided

date calendar = NEW GregorianCalendar(newYear, newMonth - 1, newDay)

// Retrieve the current item using the provided hash and linked list pointers

currentItem = itemHashTable.search(hashPointer, linkedListPointer)

// Update the attributes of the current item currentItem.name = newName

currentItem.description = newDescription currentItem.dateLimit = calendar

// Retrieve the updated item using the same hash and linked list pointers

newItem = itemHashTable.search(hashPointer, linkedListPointer)

END PROCEDURE

**PROCEDURE deleteItem**(hashPointer: INTEGER, linkedListPointer: INTEGER)

// Retrieve the current item using the provided hash and linked list pointers

currentItem = itemHashTable.search(hashPointer, linkedListPointer)

// Remove the current item from the itemHashTable using its name as the key itemHashTable.removeElement(currentItem.getName(), currentItem)

// Create an action of type 2 (e.g., a deletion action) for the current item

createAction(2, currentItem)

END PROCEDURE

**HashTable Main Methods**

**FUNCTION addElement**(key: K, value: V) RETURNS BOOLEAN

placement = 0

IF key is an instance of String THEN

placement = stringSlotFinder(cast key to String)

hashTable[placement].addLast(value)

RETURN TRUE

ELSE IF key is an instance of Integer THEN

IF cast key to Integer > 0 THEN

placement = kModM(cast key to Integer, length of hashTable)

hashTable[placement].addLast(value)

RETURN TRUE

END IF

ELSE IF key is an instance of Double THEN

IF cast key to Double > 0 AND cast key to Double < 1 THEN

placement = kM(cast key to Double, length of hashTable)

hashTable[placement].addLast(value)

RETURN TRUE

END IF

END IF

// If none of the conditions matched, return FALSE

RETURN FALSE

END FUNCTION

**FUNCTION search**(keyMajor: INTEGER, keyMinor: INTEGER) RETURNS V

// Subtract 1 from keyMajor to account for 0-based indexing

adjustedKeyMajor = keyMajor - 1

// Search the hashTable at the adjusted keyMajor with keyMinor

result = hashTable[adjustedKeyMajor].searchWithId(keyMinor)

// Return the result

RETURN result

END FUNCTION

**Priority Queue Main Methods**

**PROCEDURE offer**(element: T)

// Add the element to the heap

heap.add(element)

// Get the current index of the newly added element

currentIndex = size of heap - 1

// Continue adjusting the heap by comparing the element with its parent

WHILE currentIndex > 0 DO

// Calculate the index of the parent element

parentIndex = (currentIndex - 1) / 2

// If the element is greater than or equal to its parent, exit the loop

IF comparator.compare(heap[currentIndex], heap[parentIndex]) >= 0 THEN BREAK END IF

// Swap the current element with its parent

swap(currentIndex, parentIndex)

// Update the current index to the parent index for the next iteration

currentIndex = parentIndex

END WHILE

END PROCEDURE

**FUNCTION poll**() RETURNS T

// Check if the priority queue is empty

IF isEmpty() THEN

RAISE IllegalStateException("PriorityQueue is empty")

END IF

// Get the root element (smallest element) from the heap

root = heap.get(0)

// Get the index of the last element in the heap

lastIndex = size of heap - 1

// Replace the root with the last element and remove the last element

heap.set(0, heap.get(lastIndex))

heap.remove(lastIndex)

// Start adjusting the heap to maintain the heap property

currentIndex = 0

WHILE TRUE DO

// Calculate indices for the left and right child

leftChildIndex = 2 \* currentIndex + 1

rightChildIndex = 2 \* currentIndex + 2

smallestChildIndex = currentIndex

// Check if the left child is smaller than the current smallest

IF leftChildIndex < size of heap AND comparator.compare(heap.get(leftChildIndex), heap.get(smallestChildIndex)) < 0 THEN

smallestChildIndex = leftChildIndex

END IF

// Check if the right child is smaller than the current smallest

IF rightChildIndex < size of heap AND comparator.compare(heap.get(rightChildIndex), heap.get(smallestChildIndex)) < 0 THEN

smallestChildIndex = rightChildIndex

END IF

// If the current element is the smallest, exit the loop

IF smallestChildIndex = currentIndex THEN

BREAK

END IF

// Swap the current element with the smallest child

swap(currentIndex, smallestChildIndex)

// Update the current index to the smallest child for the next iteration

currentIndex = smallestChildIndex

END WHILE

// Return the root element

RETURN root

END FUNCTION

**Stack Main Methods**

**PROCEDURE push**(data: T)

// Create a new doubly-linked node with the given data and position

newNode = NEW NodeDouble(data, size)

// Set the previous node of the new node to the current top node

newNode.setPrev(top)

// Update the top node to the new node

top = newNode

// Increment the size of the stack

size++

END PROCEDURE

**FUNCTION pop**() RETURNS T

// Check if the stack is empty

IF isEmpty() THEN

// Return null if the stack is empty

RETURN NULL

END IF

// Get the content of the top node (element to be removed)

temp = top.getContent()

// Update the top of the stack to the previous node

top = CAST(top.getPrev() TO NodeDouble<T>)

// Decrement the size of the stack

size--

// Return the removed element

RETURN temp

END FUNCTION