In the util package, we are planning to have the next classes, implemented using Java :

Class: DoubleLinkedList<T>

- first: DoubleNode<T>

- last: DoubleNode<T>

- numberOfElements: int

Constructor DoubleLinkedList():

first = null

last = null

numberOfElements = 0

Method: isEmpty() -> bool

Return (first == null)

Method: addFirst(value: T)

newNode = Create a new DoubleNode with value

If first is null:

first = newNode

last = newNode

Else:

Set newNode's next to first

Set first's previous to newNode

Set first to newNode

Increment numberOfElements by 1

Method: addLast(value: T)

newNode = Create a new DoubleNode with value

If last is null:

first = newNode

last = newNode

Else:

Set newNode's previous to last

Set last's next to newNode

Set last to newNode

Increment numberOfElements by 1

Method: findNode(index: int) -> DoubleNode<T>

currentNode = first

If index is between 0 and numberOfElements:

localIndex = 0

While localIndex < index:

Set currentNode to currentNode's next

Increment localIndex by 1

Return currentNode

Method: delete(value: T)

currentNode = first

While currentNode is not null:

If currentNode's value equals value:

If currentNode is first:

Set first to currentNode's next

If first is not null:

Set first's previous to null

Else:

Set last to null

Else If currentNode is last:

Set last to currentNode's previous

If last is not null:

Set last's next to null

Else:

Set first to null

Else:

Set previous to currentNode's previous

Set next to currentNode's next

Set previous's next to next

Set next's previous to previous

Decrement numberOfElements by 1

Return

Set currentNode to currentNode's next

Method: deleteFirst() -> T

If first is not null:

deletedValue = first's value

If first is last:

Set first to null

Set last to null

Else:

Set first to first's next

Set first's previous to null

Decrement numberOfElements by 1

Return deletedValue

Return null

Method: deleteLast() -> T

If last is not null:

deletedValue = last's value

If first is last:

Set first to null

Set last to null

Else:

Set last to last's previous

Set last's next to null

Decrement numberOfElements by 1

Return deletedValue

Return null

Method: modifyContent(index: int, value: T)

If index is between 0 and numberOfElements:

currentNode = first

localIndex = 0

While localIndex < index:

Set currentNode to currentNode's next

Increment localIndex by 1

If currentNode is not null:

Set currentNode's value to value

Method: toString() -> string

message = ""

currentNode = first

While currentNode is not null:

Append currentNode's string representation to message

Set currentNode to currentNode's next

Return message

Method: getNumberOfElements() -> int

Return numberOfElements

Method: getFirst() -> DoubleNode<T>

Return first

Method: setFirst(first: DoubleNode<T>)

Set this.first to first

Method: getLast() -> DoubleNode<T>

Return last

Method: setLast(last: DoubleNode<T>)

Set this.last to last

Class: DoubleNode<T>

- value: T

- next: DoubleNode<T>

- previous: DoubleNode<T>

Constructor DoubleNode(value: T):

Set this.value to value

Set this.next to null

Set this.previous to null

Method: getValue() -> T

Return value

Method: setValue(value: T)

Set this.value to value

Method: getNext() -> DoubleNode<T>

Return next

Method: setNext(next: DoubleNode<T>)

Set this.next to next

Method: getPrevious() -> DoubleNode<T>

Return previous

Method: setPrevious(previous: DoubleNode<T>)

Set this.previous to previous

Method: toString() -> string

Return value converted to string

Class: HashNode<K, V>

- key: K

- value: V

- next: HashNode<K, V>

- previous: HashNode<K, V>

- status: HashNodeStatus

Constructor HashNode(key: K, value: V):

Set this.key to key

Set this.value to value

Set this.next to null

Set this.previous to null

Set this.status to HashNodeStatus.ACTIVE

Method: getKey() -> K

Return key

Method: setKey(key: K)

Set this.key to key

Method: getValue() -> V

Return value

Method: setValue(value: V)

Set this.value to value

Method: getNext() -> HashNode<K, V>

Return next

Method: setNext(next: HashNode<K, V>)

Set this.next to next

Method: getPrevious() -> HashNode<K, V>

Return previous

Method: setPrevious(prev: HashNode<K, V>)

Set this.previous to prev

Method: add(nextNode: HashNode<K, V>)

If next is null:

Set next to nextNode

Else:

Call add on next with nextNode as an argument

Method: removeElement(key: K)

If next.getKey() equals key:

Set next's status to HashNodeStatus.DELETED

Else:

If next.getNext() is not null:

Call removeElement on next.getNext() with key as an argument

Method: restoreElement(key: K) -> bool

If next.getKey() equals key:

Set next's status to HashNodeStatus.ACTIVE

Return true

Else:

If next.getNext() is not null:

Return restoreElement on next.getNext() with key as an argument

Return false

Method: getObject(key: K) -> HashNode<K, V>

If next is not null:

If next.key compareTo key equals 0:

If next's status is HashNodeStatus.ACTIVE:

Return next

Else:

Raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else:

Return getObject on next with key as an argument

Else:

Raise NonExistentKeyException with "The object with the key provided doesn't exist"

Method: getStatus() -> HashNodeStatus

Return status

Method: setStatus(status: HashNodeStatus)

Set this.status to status

Method: print() -> string

msg = ""

If status is HashNodeStatus.ACTIVE:

If value is not null:

Append "Key: " + key + ", " + value converted to string to msg

If next is not null:

Append next.print() to msg

Return msg

Class: HashTable<K, V>

- HASH\_SIZE: int

- hashList: Array of HashNode<K, V>

- size: int

Constructor HashTable():

Set this.size to 0

Create an empty array hashList of size HASH\_SIZE

Method: hashFunction(key: K) -> int

Calculate the hash code of the key

Calculate the index using floorMod with HASH\_SIZE

Return the index

Method: isEmpty() -> bool

If size is not 0, return false; otherwise, return true

Method: getSize() -> int

Return size

Method: insertElement(key: K, value: V)

Calculate the index using hashFunction with key

Create a new HashNode<K, V> newNode with key and value

If hashList at index is not null:

Call add on hashList[index] with newNode as an argument

Else:

Set hashList at index to newNode

Increment size by 1

Method: searchElement(key: K) -> HashNode<K, V>

Calculate the index using hashFunction with key

If isEmpty is true, raise HashIsEmptyException with "The hash table is empty"

Else:

If hashList at index is null, raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else If hashList at index's key compareTo key equals 0:

If hashList at index's status is HashNodeStatus.ACTIVE:

Return hashList at index

Else, raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else, return getObject on hashList at index with key as an argument

Method: deleteElement(key: K)

Calculate the index using hashFunction with key

If isEmpty is true, raise HashIsEmptyException with "The hash table is empty"

Else:

If hashList at index is null, raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else:

If hashList at index's next is null:

Set hashList at index to null

Decrement size by 1

Else, call removeElement on hashList at index with key as an argument

Method: restoreElement(key: K, value: V)

Calculate the index using hashFunction with key

If hashList at index is null:

Create a new HashNode<K, V> with key and value and set it at hashList[index]

Else:

If hashList at index's next is null:

If hashList at index's key equals key:

Set hashList at index's status to HashNodeStatus.ACTIVE

Else, add a new HashNode<K, V> with key and value to hashList at index

Else:

If restoreElement on hashList at index with key as an argument is false:

Add a new HashNode<K, V> with key and value to hashList at index

Method: print() -> string

msg = ""

For each index in the range 0 to the length of hashList:

If hashList at index is not null:

If hashList at index's next is null:

If hashList at index's status is HashNodeStatus.ACTIVE:

Append "Key: " + key + ", " + value converted to string to msg

Else, append the result of calling print on hashList at index to msg

Return msg

Class: MaxHeap<T extends Comparable<T>>

- size: int

- heap: Array of T

Constructor MaxHeap(maxsize: int)

Initialize this.heap as a new array of Comparable with maxsize

Set this.size to 0

Method: maxHeapify(i: int)

l = leftChild(i)

r = rightChild(i)

largest = i

If l is less than size and heap[l] is greater than heap[i], set largest to l

If r is less than size and heap[r] is greater than heap[largest], set largest to r

If largest is not equal to i, swap(i, largest) and call maxHeapify(largest)

Method: isEmpty() -> bool

Return whether heap[0] is null

Method: getHeap() -> Array of T

Return heap

Method: getSize() -> int

Return size

Method: setSize(heapSize: int)

Set this.size to heapSize

Method: insert(element: T)

If size is greater than or equal to the length of heap, throw HeapFullException with "The heap is full"

Set heap[size] to element

current = size

While current is greater than 0 and heap[current] is greater than heap[parent(current)], swap(current, parent(current)), and set current to parent(current)

Increment size by 1

Method: extractMax() -> T

If size is less than 1, throw PriorityQueueIsEmptyException with "The heap is empty"

max = getMax()

Set heap[0] to heap[size - 1]

Set heap[size] to null

Decrement size by 1

Call maxHeapify(0)

Return max

Method: getMax() -> T

Return heap[0]

Method: swap(i: int, j: int)

temp = heap[i]

Set heap[i] to heap[j]

Set heap[j] to temp

Method: parent(index: int) -> int

Return index / 2

Method: leftChild(i: int) -> int

Return 2 \* i + 1

Method: rightChild(i: int) -> int

Return 2 \* i + 2

Method: remove(index: int)

If index is less than 0 or index is greater than or equal to size, throw ObjectNotFoundException with "The index is not valid"

If size is 1, set heap[0] to null

Else, set heap[index] to heap[size - 1], set heap[size - 1] to null, and call maxHeapify(index)

Method: getIndexForAnObject(element: T) -> int

For each i in the range 0 to the length of heap:

If heap[i] is not null and heap[i] equals element, return i

Return -1

Method: printHeap() -> string

msg = ""

For each i in range from 0 to the length of heap:

If heap[i] is not null, append heap[i] converted to string to msg

Return msg

Here's a pseudocode representation of the `Queue` class:

```plaintext

Class: Queue<T>

- list: DoubleLinkedList<T>

Constructor Queue()

Initialize this.list as a new DoubleLinkedList<T>

Method: enQueue(newElement: T)

Call addLast on list with newElement as an argument

Method: deQueue()

removedElement = Call deleteFirst on list

If removedElement is null, throw QueueIsEmptyException with "The queue is empty"

Method: front() -> DoubleNode<T>

If list is empty, throw QueueIsEmptyException with "The queue is empty"

Else, return the result of calling getFirst on list

Method: getSize() -> int

Return the result of calling getNumberOfElements on list

Method: isEmpty() -> bool

Return whether list is empty

Method: getList() -> DoubleLinkedList<T>

Return list

```

This pseudocode represents the structure and methods of the `Queue` class in a simplified, human-readable format.

Class: Stack<T>

- list: DoubleLinkedList<T>

Constructor Stack()

Initialize this.list as a new DoubleLinkedList<T>

Method: push(newElement: T)

Call addLast on list with newElement as an argument

Method: pop()

removedElement = Call deleteLast on list

If removedElement is null, throw StackIsEmptyException with "The stack is empty"

Method: top() -> DoubleNode<T>

If list is empty, throw StackIsEmptyException with "The stack is empty"

Else, return the result of calling getLast on list

Method: getSize() -> int

Return the result of calling getNumberOfElements on list

Method: isEmpty() -> bool

Return whether list is empty

In the package model, the next classes are going to be implemented :

Class: Action

- actionType: ActionType

- task: Task

- originalTask: Task

Constructor Action(actionType: ActionType, task: Task)

Set this.actionType to actionType

Set this.task to task

Constructor Action(actionType: ActionType, task: Task, originalTask: Task)

Set this.actionType to actionType

Set this.task to task

Set this.originalTask to originalTask

Method: getActionType() -> ActionType

Return actionType

Method: setActionType(actionType: ActionType)

Set this.actionType to actionType

Method: getTask() -> Task

Return task

Method: setTask(task: Task)

Set this.task to task

Method: getOriginalTask() -> Task

Return originalTask

Method: setOriginalTask(originalTask: Task)

Set this.originalTask to originalTask

Class: Controller

- hashTableTask: HashTable<Integer, Task>

- actions: Stack<Action>

- queueTask: Queue<Task>

- heapTask: MaxHeap<Task>

Constructor Controller()

Initialize hashTableTask as a new HashTable<Integer, Task>

Initialize actions as a new Stack<Action>

Initialize queueTask as a new Queue<Task>

Initialize heapTask as a new MaxHeap<Task> with a max size of 200

Method: addTask(name: String, description: String, strLimitDate: String, priorityLevel: int, key: int) throws HeapFullException

Split strLimitDate by "/"

Convert parts to year, month, and day

Create a new GregorianCalendar instance with year, month-1, and day

Set priority based on priorityLevel

Create a new Task with name, description, key, limitDate, and priority

If priority is PRIORITY

Insert newTask into heapTask

Else if priority is NON\_PRIORITY

Enqueue newTask into queueTask

Insert newTask into hashTableTask

Create a new Action of type ADD with newTask and push it onto actions

Method: modifyTask(key: int, newName: String, newDescription: String, newStrLimitDate: String, newPriorityLevel: int) throws HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException, CloneNotSupportedException

Retrieve task from hashTableTask using key

Clone task to originalTask

Split newStrLimitDate by "/"

Convert parts to year, month, and day

Create a new GregorianCalendar instance with year, month-1, and day

Set newPriority based on newPriorityLevel

Update task with newName, newDescription, newLimitDate, and newPriority

If currentPriority is not equal to newPriority

If currentPriority is PRIORITY

Find and remove task from heapTask

Enqueue task into queueTask

Else if currentPriority is NON\_PRIORITY

Delete task from queueTask

Insert task into heapTask

Create a new Action of type MODIFY with task and originalTask, and push it onto actions

Method: showAllTasks() -> String

Return the result of calling print on hashTableTask

Method: showPrioritaryTasks() -> String

Return the result of calling printHeap on heapTask

Method: showNonPrioritaryTasks() -> String

Return the result of calling toString on queueTask.getList()

Method: showFirstPrioritaryTask() -> String throws PriorityQueueIsEmptyException

Return the result of calling toString on getMax of heapTask

Method: showFirstNonPrioritaryTask() -> String throws QueueIsEmptyException

Return the result of calling toString on front of queueTask

Method: managePriorityTask() throws PriorityQueueIsEmptyException, HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, QueueIsEmptyException

Extract the maximum task from heapTask

Get the key of the currentTask

Retrieve the hashNode from hashTableTask using the key

Set the status of the hashNode to DELETED

Create a new Action of type COMPLETE with the currentTask and push it onto actions

Method: manageNonPriorityTask() throws QueueIsEmptyException, HashIsEmptyException, NonExistentKeyException

Get the front task from queueTask

Get the key of the currentTask

Retrieve the hashNode from hashTableTask using the key

Set the status of the hashNode to DELETED

Dequeue the currentTask from queueTask

Create a new Action of type COMPLETE with the currentTask and push it onto actions

Method: revertLastAction() throws StackIsEmptyException, HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException

Get the last action from actions

Get the task from the last action

If the last action is of type ADD

If task is PRIORITY

Find and remove task from heapTask

Else if task is NON\_PRIORITY

Delete task from queueTask

Delete task from hashTableTask

Else if the last action is of type MODIFY

Get the originalTask from the last action

Set the name, description, limitDate, and priority of task to match originalTask

Else (the last action is of type COMPLETE)

If task is PRIORITY

Insert task into heapTask

Else if task is NON\_PRIORITY

Enqueue task into queueTask

Restore task in hashTableTask using key

Method: getHeapSize() -> int

Return the size of heapTask

Class: Task

- name: String

- description: String

- key: int

- limitDate: Calendar

- priorityLevel: PriorityLevel

Constructor Task(name: String, description: String, key: int, limitDate: Calendar, priorityLevel: PriorityLevel)

Set name to the provided name

Set description to the provided description

Set key to the provided key

Set limitDate to the provided limitDate

Set priorityLevel to the provided priorityLevel

Method: getName() -> String

Return the name

Method: setName(name: String)

Set the name to the provided name

Method: getDescription() -> String

Return the description

Method: setDescription(description: String)

Set the description to the provided description

Method: getLimitDate() -> Calendar

Return the limitDate

Method: setLimitDate(limitDate: Calendar)

Set the limitDate to the provided limitDate

Method: getLimitDateString() -> String

Create a SimpleDateFormat instance with the format "yyyy/MM/dd"

Format the limitDate using the SimpleDateFormat

Return the formatted date as a string

Method: getPriorityLevel() -> PriorityLevel

Return the priorityLevel

Method: setPriorityLevel(priorityLevel: PriorityLevel)

Set the priorityLevel to the provided priorityLevel

Method: compareTo(otherTask: Task) -> int

Compare the limitDate of this task with the limitDate of otherTask

Return the result of the comparison (negative if this task's date is earlier, positive if it's later)

Method: toString() -> String

Return a string with the task's name, description, formatted limitDate, and priorityLevel

Method: getKey() -> int

Return the key

Method: clone() -> Object

Call the superclass's clone method and return the result

And finally, the package ui will have the class main, which is going to have the next implementation :

Class: Main

- sc: Scanner

- controller: Controller

Constructor Main()

Initialize sc with a new Scanner reading from System.in

Initialize controller with a new Controller

Method: main(String[] args)

Create a new Main instance (main)

Initialize option with -1

Loop (while option is not 0)

Call showMenuAndGetOption() to get userOption

Call answerOption(userOption) to handle the user's choice

End Loop

Method: answerOption(int userOption)

Switch on userOption

Case 0:

Print "¡Goodbye!"

Break

Case 1:

Call addTask()

Break

Case 2:

Call modifyTask()

Break

Case 3:

Call showAllTasks()

Break

Case 4:

Call managePriorityTask()

Break

Case 5:

Call manageNonPriorityTask()

Break

Case 6:

Call revertLastAction()

Break

Case 9:

Call testCases()

Break

End Switch

Method: showMenuAndGetOption() -> int

Print the main menu

Read an integer from the user and assign it to input

Return input

Method: addTask()

Print prompts and read user input for task details (title, description, date, state, key)

Try

Call controller.addTask(title, description, date, state, key)

Catch HeapFullException e

Print e.getMessage()

Method: modifyTask()

Read user input for the key of the task to modify

Read user input for the new task details (title, description, date, state)

Try

Call controller.modifyTask(key, title, description, date, state)

Catch HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException, CloneNotSupportedException e

Print e.getMessage()

Print "The task was added successfully"

Method: showAllTasks()

Call controller.showAllTasks()

Print the result

Method: managePriorityTask()

Try

If controller.showPrioritaryTasks() is not empty

If controller.getHeapSize() is not 0

Print "This is the priority task with the nearest due date"

Print controller.showFirstPrioritaryTask()

Print "Would you like to mark this task as completed?"

Read option

If option is 1

Try

Call controller.managePriorityTask()

Catch ObjectNotFoundException, QueueIsEmptyException e

Print e.getMessage()

End Try

End If

Else

Print "There aren't any pending prioritary tasks to manage"

End If

Else

Print "There aren't any pending prioritary tasks to manage"

End If

Catch PriorityQueueIsEmptyException, HashIsEmptyException, NonExistentKeyException e

Print e.getMessage()

Print "The task was successfully completed"

Method: manageNonPriorityTask()

Try

If controller.showNonPrioritaryTasks() is not empty

Print "This is the first non-priority task registered"

Print controller.showFirstNonPrioritaryTask()

Print "Would you like to mark this task as completed?"

Read option

If option is 1

Try

Call controller.manageNonPriorityTask()

Catch HashIsEmptyException, NonExistentKeyException e

Print e.getMessage()

End Try

End If

Else

Print "There aren't any pending non-priority tasks to manage"

End If

Catch QueueIsEmptyException e

Print e.getMessage()

Method: revertLastAction()

Try

Call controller.revertLastAction()

Catch StackIsEmptyException, HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException e

Print e.getMessage()

Method: testCases()

Try

Call controller.addTask("name", "null", "2022/11/01", 1, 12)

Call controller.addTask("name2", "null", "2022/11/02", 1, 1)

Call controller.addTask("name3", "null", "2022/11/03", 2, 15)

Call controller.addTask("name4", "null", "2022/11/04", 2, 18)

Catch HeapFullException e

Print e.getMessage()