**Tarea Integradora I**

Gloria Vanesa Vicuña - A00369332

Ricardo Medina Sterling - A00369009

Alejandro Osejo Ochoa - A00372469

Computación y Estructuras Discretas I

Universidad Icesi

2023

**DESIGN**

1. TAD design of the data structures used.

|  |
| --- |
| **HashTable** |
| **HashTable= {Size = <size>, Table = <lista(size)>** |
| {inv: <∃! key ∈ Table>} |
| Primitive Operations   * Hash: <key> → <hash> * Put: <key>, <value> → Table * Get: <key> → value * Remove: <key> → value * Size:    → size |

|  |
| --- |
| hash(key)  “Allows you to retrieve the Hash Code with the given key”  {pre:(key.hashCode() mod size) < 0}  {hash = (key.hashCode() mod size) + size} |

|  |
| --- |
| put(key,value)  “Inserts a key-value pair into the hash table.”  {pre: (table.key.hash.code() mod size) ≠ ∅ }  {post: key.hash, new HashNode(key, value)} |

|  |
| --- |
| get(key)  “Returns the value associated with the given key.”  {pre: (table.key.hash.code() mod size) ≠ ∅ }  {post: table.get(hash).getValue} |

|  |
| --- |
| remove (key)  “Removes the key-value pair associated with the given key.”  {pre: (table.key.hash.code() mod size) ≠ ∅ }  {post: table.get(hash).getValue  ∧  hash(k) ∉ table} |

|  |
| --- |
| **MaxPriorityQueue** |
| **MaxPriorityQueue = {Size = <0>, heap= <HeapNode(capacity)>** |
| {inv: <Invariante del TAD>} |
| Primitive Operations   * GetMax :     → <max> * ExtracMax:    → <max> * MaxInset: <key>, <value>  → <heap> * Parent: <i> →  <parent> * Swap: <i>, <j> →  <j>, <i> * MaxHeapify: <i> → <maxHeapify> |

|  |
| --- |
| getMax()  “Returns the element with the highest priority in the priority queue.”  {pre: (heap[0].getValue) ≠ ∅ }  {post:heap[0].getValue } |

|  |
| --- |
| extractMax()  “returns and removes the element with the highest priority in the priority queue”  {pre: (heap[0].getValue) ≠ ∅ }  {post: max ∉ heap } |

|  |
| --- |
| maxInsert(key, value)  “inserts an element into the priority queue with the given key and value.”  {pre: (size ≠ heap.length }  {post: heap[size]=HeapNode<>(key, value) ∧ size= size+1} |

|  |
| --- |
| parent(i)  “calculates the parent”  {pre: i ∉ Z }  {post: (i-1)/2} |

|  |
| --- |
| swap(i, j)  “exchange the position of 2 elements”  {pre: i ∈ Z ∧ j ∈ Z }  {post:} |
| maxHeapify(i)  “reorders elements in the heap to maintain the highest priority of the heap”  {pre: i ∈ Z}  {post: } |

|  |
| --- |
| isEmpty()  “check if priority queue is empty”  {pre:  }  {post: } |

|  |
| --- |
| **MinPriorityQueue** |
| **MinPriorityQueue = {Size = <0>, heap= <HeapNode(capacity)>** |
| {inv: <Invariante del TAD>} |
| Primitive Operations   * GetMin:     → <min> * ExtracMax:    → <min> * MinInset: <key>, <value>  → <heap> * Parent: <i> →  <parent> * Swap: <i>, <j> →  <j>, <i> * MinHeapify: <i> → <minHeapify> * isEmpty:   → <True> ∨ <False> |

|  |
| --- |
| getMin()  “returns the element with the minor priority in the priority queue ”  {pre: (heap[0].getValue) ≠ ∅ }  {post:heap[0].getValue } |

|  |
| --- |
| extractMin()  “returns and removes the element with the minor priority in the priority queue”  {pre: (heap[0].getValue) ≠ ∅ }  {post: max ∉ heap } |

|  |
| --- |
| minInsert(key, value)  “inserts an element into the priority queue with the given key and value.”  {pre: (size ≠ heap.length }  {post: heap[size]=HeapNode<>(key, value) ∧ size= size+1} |
| parent(i)  “calculates the parent”  {pre: i ∉ Z }  {post: (i-1)/2} |

|  |
| --- |
| swap(i,j)  “exchange the position of 2 elements”  {pre: i ∈  Z ∧ j ∈ Z }  {post:} |

|  |
| --- |
| minHeapify(i)  “reorders heap elements to maintain the lowest heap priority”  {pre: i ∈ Z }  {post: } |

|  |
| --- |
| isEmpty()  “check if priority queue is empty”  {pre:  }  {post: } |

|  |
| --- |
| **Queue** |
| **Queue = {Size = <0>, queue= Object[length]** |
| {inv: <Invariante del TAD>} |
| Primitive Operations   * Enqueue: <data> → <queue> * Dequeue:  → <data> * Peek:  → <queue> * isEmpty:   → <True> ∨ <False> * size: → <size> * getQueue: → <queue> * setQueue: <queue>, <size> → <queue>, <size> |

|  |
| --- |
| enqueue(data)  “inserts an element at the end of the queue”  {pre: size ≠ queue.length}  {post:queue[size] = data ∧ size= size+1} |

|  |
| --- |
| dequeue()  “removes and returns the element from the beginning of the queue”  {pre: isEmpty = false}  {post: queue[i] = queue[i +1] ∧ size= size-1} |

|  |
| --- |
| peek(data)  “returns the element from the beginning of the queue without removing it”  {pre: isEmpty = false}  {post: queue[0]} |

|  |
| --- |
| isEmpty()  “check if priority queue is empty”  {pre: }  {post: } |

|  |
| --- |
| size()  “returns the size of the queue”  {pre: }  {post:} |

|  |
| --- |
| getQueue()  “show the queue”  {pre: }  {post:} |

|  |
| --- |
| setQueue(queue)  “gives a new value to the queue and a size”  {pre: }  {post:} |