|  |  |  |
| --- | --- | --- |
| TAD Hashtable | | |
| Hashtable = {table: < List₁, List₂, List₃, …, Listₙ>}   * Where:   List = {elements: <Node₁, Node₂, Node₃, …, Nodeₙ>}   * Where:   Node = {key: <key>, value: <value>} | | |
| { inv: Node.key ∈ Comparable } | | |
| Primitive functions: | | |
| * CreateHashtable: |  | * HashTable |
| * Put: | Hashtable X Value X Key | * Hashtable |
| * Get: | Hashtable X Key | * Value |
| * Remove: | Hashtable X Key | * Boolean |
| * Hash: | Hashtable X Key | * Integer |

CreateHashtable()

“Creates a new Hashtable object with 0 elements”

{ pre: TRUE}

{ post: hasht = {table: [ ] } }

Put(hasht,key,value)

“Adds a new element to one of the lists of the table”

{ pre: hasht = { table = […] }

{ post: hasht = { table = […, listₖ = [ …, Nodeₘ = key: <key>, value: <value>, …], …]}

Remove(hasht,key)

“Deletes an element from one of the lists of the table”.

{pre: hasht = {table = […, listₖ = [ …, Nodeₘ = key: <key>, value: <value>, …], …]}

{post: hasht = {table = […, listₖ = […], …]} True if the element was found and deleted. False otherwise.}

Get(hasht,key)

“Retrieves an element from one of the lists of the table”.

{pre: hasht = {table = […, listₖ = [ …, Nodeₘ = key: <key>, value: <value>, …], …]}

{post:

If key ∈ table: <value>

Else: Null}

Hash(hasht,key)

“Returns a list index to save a key”.

{pre: hasht = {table : […]}

{post: <Integer key’s value> % hasht.table.length}

|  |  |  |
| --- | --- | --- |
| TAD PriorityQueue | | |
| PriorityQueue = { elements: <Node₁, Node₂, Node₃, …, Nodeₙ>}   * Where:   Node = {key: <key>, value: <value>} | | |
| { inv: Node.key ∈ Comparable /\ { Nodeₙ₋₁.key >= Nodeₙ.key >= Nodeₙ₊₁.key } } | | |
| Primitive functions: | | |
| * CreatePriorityQueue: |  | * PriorityQueue |
| * Insert: | PriorityQueue X Value X Key | * PriorityQueue |
| * Extract: | PriorityQueue | * Value |

CreatePriorityQueue()

“Creates a new PriorityQueue object with 0 elements”

{ pre: TRUE}

{ post: queue = {elements: [ ] } }

Put(queue,key,value)

“Adds a new element, to elements collection, at its respective position.”

{ pre: queue = { elements = […] }

{ post: queue = { elements = […, Node : {key:<key>, value: <value>},…]} /\ { Nodeₙ₋₁.key >= Nodeₙ.key >= Nodeₙ₊₁.key } }

Extract(queue)

“Extracts the first element from the collection, returns it and deletes it”.

{pre: queue = { elements = [Node₁, Node₂, Node₃, …, Nodeₙ]}

{post: Node₁.value /\ queue = { elements = [Node₂, Node₃, …, Nodeₙ]}

|  |  |  |
| --- | --- | --- |
| TAD MaxHeap | | |
| MaxHeap = { elements: <Node₁, Node₂, Node₃, …, Nodeₙ>}   * Where:   Node = {key: <key>, value: <value>} | | |
| { inv: Node.key ∈ Comparable /\ ∀n{ Nodeₙ.key >= Node₂․ₙ.key } /\ ∀n{ Nodeₙ.key >= Node₂․ₙ₊₁.key } } | | |
| Primitive functions: | | |
| * CreateMaxHeap: |  | * MaxHeap |
| * HeapSort: | MaxHeap | * List |
| * Heapify: | MaxHeap | * MaxHeap |
| * BuildHeap: | MaxHeap X List | * MaxHeap |

Heapsort(heap)

“Returns a list with all of the element’s values inside the heap structure, in ascending order by their keys”.

{pre: heap = { elements: Node₁, Node₂, Node₃, …, Nodeₙ]} /\ ∀n{ Nodeₙ.key >= Node₂․ₙ.key } /\ ∀n{ Nodeₙ.key >= Node₂․ₙ₊₁.key } }

{post: list = [Node₁, Node₂, Node₃, …, Nodeₘ] /\ ∀m{ Nodeₘ.key <= Nodeₘ₊₁.key } }

CreateMaxHeap()

“Creates a new CreateMaxHeap object with 0 elements”

{ pre: TRUE}

{ post: heap = {elements: [ ] } }

Heapify(heap,n)

“Orders a specified branch of elements”.

{ pre: heap = { elements = […, Nodeₙ, …, Node₂․ₙ, Node₂․ₙ₊₁, …] } /\ { (Nodeₙ.key < Node₂․ₙ.key) \/ (Nodeₙ.key < Node₂․ₙ₊₁.key) } }

{ post: heap = { elements = […, Nodeₙ, …, Node₂․ₙ, Node₂․ₙ₊₁, …] } /\ { Nodeₙ.key >= Node₂․ₙ.key } /\ ∀n{ Nodeₙ.key >= Node₂․ₙ₊₁.key } }

BuildHeap(heap,list)

“Builds a MaxHeap structure based on the given list”.

{pre: heap = { elements : […]} /\ list = [Node₁, Node₂, Node₃, …, Nodeₙ] }

{post: heap = { elements: [Node₁, Node₂, Node₃, …, Nodeₘ]} /\ ∀m{Nodeₘ.key >= Node₂․ₘ.key } /\ ∀m{ Nodeₘ.key >= Node₂․ₘ₊₁.key }}