***TAD Hashtable***

| ***TAD Hash Table*** | | |
| --- | --- | --- |
| Hashtable={ key= <K>, value= <T>} | | |
| {inv: size ≠ 0} | | |
| **Primitive Operations** | | |
| ***InsertElement*** | **Hashtable x K x T → *Entero*** | |
| ***SearchElement*** | ***Hashtable x K x T → Entero*** | |
| ***DeleteElement*** | ***Hashtable x K, T → Hashtable*** | |

| ***InsertElement(K ,T)*** | |
| --- | --- |
| ***“This method allow us to store a key on the hashtable”*** | |
| ***{pre: }*** | |
| ***{post; }*** | |

| ***searchElement(K ,T)*** | |
| --- | --- |
| ***“This method allow us to search a stored key on the hashtable”*** | |
| ***{pre: }*** | |
| ***{post; }*** | |

| ***deletetElement(K ,T)*** | |
| --- | --- |
| ***“This method allow us to delete a key stored on the hashtable”*** | |
| ***{pre: }*** | |
| ***{post; }*** | |

***TAD HASHNODE***

***TAD Stack***

| ***TAD Stack*** | | |
| --- | --- | --- |
| Stack = {{e1,e2, e3, ..., en}, top} | | |
| {inv: 0 ≤ n ∧ Size(Stack) = n ∧ top = en} | | |
| **Primitive Operations** | | |
| ***stack*** | **Stack → Stack** | |
| ***isEmpty*** | **Stack → *boolean*** | |
| ***top*** | ***Stack → Element*** | |
| ***pop*** | ***Stack → Stack*** | |
| ***push*** | ***Stack x Element → Stack*** | |

| ***stack()*** | |
| --- | --- |
| *“This method creates an empty stack”* | |
| *{pre: -}* | |
| *{post: Stack s = ∅ }* | |

| ***push(Element e)*** | |
| --- | --- |
| *“This method add a new element to the stack”* | |
| *{pre: Stack s {isEmpty = false} \/ s=∅ }* | |
| *{post: Stack s= e1, e2.. en, e} ∨ s={e}* | |

| ***pop()*** | |
| --- | --- |
| *“Extract the most recent inserted element”* | |
| *{pre: Stack s ≠ ∅ , s={e1, e2, …en} }* | |
| *{post: s={e1,e2… en-1} }* | |

| ***top()*** | |
| --- | --- |
| *“Recover the value of the element of the top of the stack s”* | |
| *{pre: Stack s ≠ ∅, s={e1,e2,...en}}* | |
| *{post: Element en}* | |

| ***isEmpty()*** | |
| --- | --- |
| *“Represent if the stack s is empty or not”* | |
| *{pre: Stack s }* | |
| *{post; true if s ≠∅, False if s≠∅ }* | |

***TAD Queue***

| ***TAD Queue*** | | |
| --- | --- | --- |
| Queue= {{e1,e2, e3, ..., en}, front, back} | | |
| {inv: 0 ≤ n ∧ Size(Queue) = n ∧ front = e1 ∧ back = en} | | |
| **Primitive Operations** | | |
| ***Queue*** | **Queue→ Queue** | |
| ***isEmpty*** | **Queue→ *boolean*** | |
| ***dequeue*** | ***Queue → Element*** | |
| ***enqueue*** | ***Queue x Element →* Queue** | |
| ***front*** | ***Queue →* Element** | |

| ***Queue()*** | |
| --- | --- |
| *“This method creates an empty Queue q”* | |
| *{pre: -}* | |
| *{post: Queue q = ∅ }* | |

| ***isEmpty()*** | |
| --- | --- |
| *“Represent if the stack s is empty or not”* | |
| *{pre: Queue q}* | |
| *{post: true if s ≠∅, False if s≠∅ }* | |

| ***enqueue(Element e)*** | |
| --- | --- |
| *“Add a new element to the back of the queue q”* | |
| *{pre: Queue q= {e1,e2… en} or q≠∅ }* | |
| *{post: Queue q= {e1,e2…en, e} or q={e}}* | |

| ***dequeue()*** | |
| --- | --- |
| *“Extract the element of the queue front”* | |
| *{pre: Queue q≠∅ or q={e1,e2… en}}* | |
| *{post: Queue q= {e2…en, en-1} and element e1* | |

| ***front()*** | |
| --- | --- |
| *“Recover the value e at the front of the Queue q”* | |
| *{pre: Queue q≠∅, q= {e1,e2… en}}* | |
| *{post: Element<T> e1}* | |

***TAD Priority Queue***

| ***TAD PriorityQueue*** | | |
| --- | --- | --- |
| PriorityQueue= {{e1,e2, e3, ..., en}, heap} | | |
| {inv: 0 ≤ n ∧ Size(***PriorityQueue***)} | | |
| **Primitive Operations** | | |
| ***PriorityQueue*** | ***PriorityQueue→ PriorityQueue*** | |
| ***isEmpty*** | ***PriorityQueue→boolean*** | |
| ***insert*** | ***PriorityQueue x T → PriorityQueue*** | |
| ***getMax*** | ***PriorityQueue→ T*** | |
| ***deleteMax*** | ***PriorityQueue→ PriorityQueue*** | |
| ***size*** | ***PriorityQueue→PriorityQueue//NOSE LAS SALIDAS*** | |

| ***PriorityQueue()*** | |
| --- | --- |
| *“This method construct a new empty priority Queue”* | |
| *{pre:* ***-****}* | |
| *{post:PriorityQueue q=∅}* | |

| ***isEmpty()*** | |
| --- | --- |
| *“Represent if the priority Queue is empty or not”* | |
| *{pre: PriorityQueue p}* | |
| *{post: true if p ≠∅, False if p≠∅ }* | |

| ***insert( T )*** | |
| --- | --- |
| *“Add a new element z to the priority Queue p”* | |
| *{pre: PriorityQueue p=∅ or p={e1,e2,...en}}* | |
| *{post: p={e1,e2,en, z } or p={z}}* | |

| ***deleteMax( )*** | |
| --- | --- |
| *“deletes the maximum element of the priority Queue p”* | |
| *{pre: PriorityQueue p=∅ or p={e1,e2,...en}}* | |
| *{post: p={e1,e2,en, z } or p={z}}* | |

| ***getMax( )*** | |
| --- | --- |
| *“return the maximum element of the priority Queue p”* | |
| *{pre: PriorityQueue p=∅ or p={e1,e2,...en}}* | |
| *{post: Element e1}* | |

***TAD Double Linked List***

| ***TAD DoubleList*** | | |
| --- | --- | --- |
| {DoubleList d={n1,n2,n3..ni}, first, last, numElements} | | |
| {int size(DoubleList)≥0} | | |
| **Primitive Operations** | | |
| ***DoubleList*** | **DoubleList→ DoubleList** | |
| ***isEmpty*** | **DoubleList → *boolean*** | |
| ***addFirst*** | **DoubleList x Element<T>*→* DoubleList** | |
| ***addLast*** | **DoubleList x Element<T> → DoubleList** | |
| ***deleteFirst*** | **DoubleList *→* Element<T>** | |
| ***deleteLast*** | **DoubleList *→* Element<T>** | |
| ***findNode*** | **DoubleList x Integer → DoubleNode<T>** | |
| ***modifyContent*** | **DoubleList x Integer x T → DoubleList** | |

| **DoubleList *()*** | |
| --- | --- |
| *“This method construct a new empty* Double Linked List *”* | |
| *{pre:* ***-****}* | |
| *{post:***DoubleList**  *d=∅}* | |

| ***isEmpty()*** | |
| --- | --- |
| *“Represent if the priority double linked list is empty or not”* | |
| *{pre:* **DoubleList** *d}* | |
| *{post: true if d≠∅, False if d≠∅ }* | |

| ***addFirst( T )*** | |
| --- | --- |
| *“Add a new element z to the beginning of the double linked list”* | |
| *{pre: DoubleList d=∅ or p={e1,e2,...en}}* | |
| *{post: d={z,e1,e2,en } or d={z}}* | |

| ***addLast( T )*** | |
| --- | --- |
| *“Add a new element z to the end of the double linked list”* | |
| *{pre: DoubleList d=∅ or p={e1,e2,...en}}* | |
| *{post: d={e1,e2,en, z } or d={z}}* | |

| ***deleteFirst()*** | |
| --- | --- |
| *“Deletes the first element at the double linked list”* | |
| *{pre: DoubleList d≠∅ or d={e1,e2,...en}}* | |
| *{post: d={e2,....en, z }}* | |

| ***deleteLast( )*** | |
| --- | --- |
| *“Deletes the last element at the double linked list”* | |
| *{pre: DoubleList d≠∅ and d={e1,e2,...en}}* | |
| *{post: d={e1,e2,...en-1}}* | |

| ***modifyContent( Integer, T)*** | |
| --- | --- |
| *“Modifies the content of the element x specified in the double linked list”* | |
| *{pre: DoubleList d≠∅ and d={e1,e2,...en} and x∊ d}* | |
| *{post: d={e1,e2,...en}}* | |

| ***findNode( Integer )*** | |
| --- | --- |
| *“Search the node selected by an index”* | |
| *{pre: DoubleList p={e1,e2,...en} and x ∊ p}* | |
| *{post: Element x}* | |

***TAD DoubleNode***

| ***TAD DoubleNode*** | | |
| --- | --- | --- |
| DoubleNode= {{T value}, next, previous} | | |
| {inv: T ≠ ∅} | | |
| **Primitive Operations** | | |
| ***Queue*** | **Queue→ Queue** | |
| ***isEmpty*** | **Queue→ *boolean*** | |
| ***dequeue*** | ***Queue → Element*** | |
| ***enqueue*** | ***Queue x Element →* Queue** | |
| ***front*** | ***Queue →* Element** | |

***TAD MaxHeap***

| ***TAD MaxHeap*** | | |
| --- | --- | --- |
| ***MaxHeap***= {{e1,e2, e3, ..., en}, heap, size, maxSize} | | |
| {inv: 0 ≤ n ∧ Size(***MaxHeap***)≥0 ^ 0 ≤ maxSize ^ e1> {e2,e3,...en}} | | |
| **Primitive Operations** | | |
| ***MaxHeap*** | ***MaxHeap X Integer*→ *MaxHeap*** | |
| ***parent*** | ***MaxHeap x Integer → Integer*** | |
| ***leftChild*** | ***MaxHeap x Integer → integer*** | |
| ***rightChild*** | ***MaxHeap x Integer → integer*** | |
| ***swap*** | ***MaxHeap x Integer x Integer → MaxHeap*** | |
| ***maxHeapify*** | ***MaxHeap x Integer → MaxHeap*** | |
| ***insert*** | ***MaxHeap x T → MaxHeap*** | |
| ***remove*** | ***MaxHeap x Integer → T*** | |
| ***heapsort*** | ***MaxHeap → MaxHeap*** | |

| ***MaxHeap( Integer )*** | |
| --- | --- |
| *“Creates a Maximum heap ”* | |
| *{pre:}* | |
| *{post: MaxHeap = ⊘}* | |

| ***parent( Integer )*** | |
| --- | --- |
| *“Returns the parent of the node specified”* | |
| *{pre: MaxHeap h ≠Ø, and 0< parent < maxSize}* | |
| *{post: parent}* | |

| ***leftChild( Integer )*** | |
| --- | --- |
| *“Returns the left child of the node specified ”* | |
| *{pre: MaxHeap h ≠Ø, and 0< leftChild< maxSize}* | |
| *{post: leftChild}* | |

| ***rightChild( Integer )*** | |
| --- | --- |
| *“Returns the right child of the node specified ”* | |
| *{pre: MaxHeap h ≠Ø, and 0< leftChild< maxSize}* | |
| *{post: rightChild}* | |

| ***swap( Integer, Integer )*** | |
| --- | --- |
| *“Exchange two elements(f1,f2) given an index in the Heap ”* | |
| *{pre: f1 ៱ f2 ∊ Heap}* | |
| *{post: f1= Heap[f2]. f2= Heap[f1]}* | |

| ***maxHeapify( Integer )*** | |
| --- | --- |
| *“Exchange two elements(f1,f2) given an index in the Heap ”* | |
| *{pre: f1 ៱ f2 ∊ Heap}* | |
| *{post: f1= Heap[f2]. f2= Heap[f1]}* | |

| ***insert(T)*** | |
| --- | --- |
| *“Inserts a new element x in the heap h in the corresponding position ”* | |
| *{pre: Heap h =Ø or h= {e1,e2,...en}}* | |
| *{post: Heap h ={x} or h= {e1,e2,...en, x}}* | |

| ***remove(Integer)*** | |
| --- | --- |
| *“Removes a element x in the heap h given an index”* | |
| *{pre: Heap h =Ø or h= {e1,e2,...en,x}}* | |
| *{post:x}* | |

| ***heapSort()*** | |
| --- | --- |
| *“Inserts a new element x in the heap h in the corresponding position ”* | |
| *{pre: Heap h =Ø or h= {e1,e2,...en}}* | |
| *{post: Heap h ={x} or h= {e1,e2,...en, x}}* | |