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

| ***Hashtable()*** | |
| --- | --- |
| ***“This method allow us to create a new hashtable h”*** | |
| ***{pre: - }*** | |
| ***{post; Hashtable h}*** | |

| ***InsertElement(K ,T)*** | |
| --- | --- |
| ***“This method allow us to store a key on the hashtable h”*** | |
| ***{pre: Hashtable size ≠0 }*** | |
| ***{post; h={, , V, , ,}}*** | |

| ***searchElement(K )*** | |
| --- | --- |
| ***“This method allow us to search a stored key p on the hashtable h”*** | |
| ***{pre: Hashtable size > 0 & p∈h }*** | |
| ***{post; p<K,V>}*** | |

| ***deletetElement(K )*** | |
| --- | --- |
| ***“This method allow us to delete a key x stored on the hashtable h”*** | |
| ***{pre: Hashtable size > 0 & x∈h , h={ , , , x , ,}}*** | |
| ***{post: h={ , , , , ,} }*** | |

| ***hashFuncition(K )*** | |
| --- | --- |
| ***“This method allow us to converte a key in an index of the hashtable”*** | |
| ***{pre: K key ≠NIL}*** | |
| ***{post: index }*** | |

***TAD HASHNODE***

| ***TAD HashNode*** | | |
| --- | --- | --- |
| ***HashNode***= {K key, V value, HashNode<K, V> next, HashNode<K, V> prev} | | |
| {inv: key>0 and value ≠∅} | | |
| **Primitive Operations** | | |
| ***HashNode*** | ***HashNode* K, V→ *HashNode*** | |
| ***getKey*** | ***HashNode→K*** | |
| ***setKey*** | ***HashNode x K→ HashNode*** | |
| ***getValue*** | ***HashNode →* V** | |
| ***setValue*** | ***HashNode x V→ HashNode*** | |
| ***getNext*** | ***HashNode → HashNode<K,V>*** | |
| ***setNext*** | ***HashNode xHashNode <K,V> → HashNode*** | |
| ***getPrev*** | ***HashNode → HashNode<K,V>*** | |
| ***setPrev*** | ***HashNode x HashNode<K,V> → HashNode*** | |
| ***add*** | ***HashNode x HashNode<K,V> →HashNode*** | |
| ***removeElement*** | ***HashNode x K → HashNode*** | |
| ***getObject*** | ***HashNode x K → Hashnode<K,V>*** | |

| ***HashNode(K,V)*** | |
| --- | --- |
| *“Creates a new hash node ”* | |
| *{pre:-}* | |
| *{post: HashNode={K,V}}* | |

| ***getValue()*** | |
| --- | --- |
| *“gets the value of the hash node x”* | |
| *{pre:HashNode x ={V ≠ Ø}}* | |
| *{post: x}* | |

| ***setValue(V)*** | |
| --- | --- |
| *“modify the value of the hash node x”* | |
| *{pre:HashNode x ={V ≠ Ø}}* | |
| *{post: x}={V= new value}* | |

| ***getKey()*** | |
| --- | --- |
| *“gets the key of the hash node x”* | |
| *{pre:HashNode x ={K ≠ Ø and K≥0}}* | |
| *{post: k}* | |

| ***setKey(K)*** | |
| --- | --- |
| *“modify the key of the hash node x for a new key”* | |
| *{pre:HashNode x ={K ≠ Ø and K≥0}}* | |
| *{post: HashNode x={K= new Key}}* | |

| ***getPrev()*** | |
| --- | --- |
| *“gets the previous node of the hash node x”* | |
| *{pre: prev ≠ Ø}* | |
| *{post: prev}* | |

| ***setPrev(HashNode<K,V>)*** | |
| --- | --- |
| *“modify the previous node of the actual hash node x for a new one “* | |
| *{pre: prev ≠∅}* | |
| *{post:prev= new Node }* | |

| ***add(HashNode<K,V>)*** | |
| --- | --- |
| *“add all the next nodes when we need to chain nodes”* | |
| *{pre:-}* | |
| *{post: next= nexNode}* | |

| ***removeElement(K)*** | |
| --- | --- |
| *“remove a element x specified by a key in a chain”* | |
| *{pre:Hashnode next = x}* | |
| *{post: Hashnode next = DELETED}* | |

| ***getObject(K)*** | |
| --- | --- |
| *“Get an object with key* ***K*** *in the chaining of the hashTable h in an index”* | |
| *{pre: k ∈ h, k ≠NIL}* | |
| *{post: HashNode <k,v>}* | |

***TAD Stack***

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

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

| ***push(T 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: 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}} | | |
| {inv: 0 ≤ n ∧ Size(Queue)} | | |
| **Primitive Operations** | | |
| ***Queue*** | **Queue→ Queue** | |
| ***isEmpty*** | **Queue→ *boolean*** | |
| ***dequeue*** | ***Queue → Queue*** | |
| ***enqueue*** | ***Queue x T→* Queue** | |
| ***front*** | ***Queue →* DoubleNode<T>** | |

| ***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(T e)*** | |
| --- | --- |
| *“Add a new element e 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()*** | |
| --- | --- |
| *“deletes the element of the queue front”* | |
| *{pre: Queue q≠∅ or q={e1,e2… en}}* | |
| *{post: Queue q= {e2,....., en}* | |

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

***TAD Priority Queue***

| ***TAD PriorityQueue*** | | |
| --- | --- | --- |
| PriorityQueue= { MaxHeap <T>heap} | | |
| {inv: 0 ≤ n ∧ Size(***PriorityQueue***)} | | |
| **Primitive Operations** | | |
| ***PriorityQueue*** | ***PriorityQueue x Integer → PriorityQueue*** | |
| ***isEmpty*** | ***PriorityQueue→boolean*** | |
| ***insert*** | ***PriorityQueue x T → PriorityQueue*** | |
| ***maximum*** | ***PriorityQueue→ T*** | |
| ***extracMax*** | ***PriorityQueue→ PriorityQueue*** | |

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

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

| ***insert( T z )*** | |
| --- | --- |
| *“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}}* | |

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

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

***TAD Double Linked List***

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

| ***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}, <DoubleNode T>next, <DoubleNode T>previous} | | |
| {inv: T ≠ ∅} | | |
| **Primitive Operations** | | |
| ***getValue*** | **DoubleNode → T** | |
| ***DoubleNode*** | **DoubleNode x T→ DoubleNode** | |
| ***setValue*** | **DoubleNode x T →DoubleNode** | |
| ***getNext*** | **DoubleNode *→* DoubleNode <T>** | |
| ***setNext*** | **DoubleNode *x* DoubleNode<T> *→* DoubleNode** | |
| ***getPrevious*** | **DoubleNode *→* DoubleNode <T>** | |
| ***setPrevious*** | **DoubleNode *x* DoubleNode<T> *→* DoubleNode** | |

| ***DoubleNode(T)*** | |
| --- | --- |
| *“Creates a new node double with content T”* | |
| *{pre:-}* | |
| *{post: DoubleNode={T}}* | |

| ***getValue()*** | |
| --- | --- |
| *“gets the value of the node”* | |
| *{pre:DoubleNode ={T ≠ Ø}}* | |
| *{post: T}* | |

| ***setValue(T)*** | |
| --- | --- |
| *“modify the value of the node, replaces the older value for the new one x”* | |
| *{pre:DoubleNode ={T ≠ Ø}}* | |
| *{post: DoubleNode={x}}* | |

| ***getNext()*** | |
| --- | --- |
| *“gets the next node of the actual node”* | |
| *{pre:DoubleNode ={T ≠ Ø} and next ≠ Ø }* | |
| *{post: next}* | |

| ***setNext(DoubleNode<T>)*** | |
| --- | --- |
| *“modify the next node of the actual node replacing the older for the new one “****n****””* | |
| *{pre:DoubleNode ={T ≠ Ø} and next ≠ Ø}* | |
| *{post: next = n}* | |

| ***getPrevious()*** | |
| --- | --- |
| *“gets the previous node of the actual node”* | |
| *{pre:DoubleNode ={T ≠ Ø} and previous≠ Ø }* | |
| *{post: previous}* | |

| ***setPrevious(DoubleNode<T>)*** | |
| --- | --- |
| *“modify the previous node of the actual node replacing the older for the new one “****n****””* | |
| *{pre:DoubleNode ={T ≠ Ø} and previous≠ Ø}* | |
| *{post: previous= n}* | |

***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*** | |
| ***buildMaxHeap()*** | ***MaxHeap → MaxHeap*** | |
| ***getIndexForanObject*** | ***MaxHeap x T → Integer*** | |

| ***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= integer/2}* | |

| ***leftChild( Integer )*** | |
| --- | --- |
| *“Returns the left child of the node specified ”* | |
| *{pre: MaxHeap h ≠Ø, and 0< leftChild< maxSize}* | |
| *{post: leftChild= integer\*2}* | |

| ***rightChild( Integer )*** | |
| --- | --- |
| *“Returns the right child of the node specified ”* | |
| *{pre: MaxHeap h ≠Ø, and 0< leftChild< maxSize}* | |
| *{post: rightChild=integer\*2+1}* | |

| ***switchPlaces( 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 )*** | |
| --- | --- |
| *“It is used to keep the property of the maxHeap after we make changes in the structure, ensure that the largest element still in the first position”* | |
| *{pre: heap ≠ NIL & heap={e1,e2,e3,…en}, e2> e1,e3…en}* | |
| *{post: large = e2}* | |

| ***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, h={e1,e2…en}}* | |

| ***heapSort()*** | |
| --- | --- |
| *“Sorts the heap from largest to smallest while maintaining the max heapify property”* | |
| *{pre: Heap h ≠Ø or h= {e1,e2,...en}}* | |
| *{post: h= {e1>e2 >e3,...>en}}* | |

| ***buildMaxHeap()*** | |
| --- | --- |
| *“Build a initial maxHeap organizing a disordered array from largest to smallest”* | |
| *{pre: Heap h ≠Ø or h= {e1,e2,...en}}* | |
| *{post: h= {e1>e2 >e3,...>en}}* | |

| ***getIndexForAnObject(T)*** | |
| --- | --- |
| *“”* | |
| *{pre: heap NIL, T∊ heap}* | |
| *{post: index}* | |