| **TAD<Hash Table>** |
| --- |
| Hash table={size,table,hashFunction,keyEqualityFunction} |
| Inv= {∀k1∀k2( k1 ∈ table(Keys) y k2 ∉ table(keys) y k1 != k2 } |
| **Main operations**  **Builder->CreationHashTable()** : Integer ->HashTable  **Modifier -> Insert():** HashTable x key x Element ->HashTable  **Modifier ->Remove():** HashTable x key x Element ->HashTable  **Analyzer -> Search():** HashTable x Key -> Element  **Analyzer -> Contains():** HashTable x key -> boolean |

| **TAD <Stack>** |
| --- |
| Stack= {push,pop, peek} |
| Inv = {Comparator(a,b)=(S={En..E3,E3,E1} S.pop=En) ∧ True } |
| **Main operations**  **Builder -> CreateStack(): ->** Stack  **Modifier -> Push():** Stack x Element -> Stack  **Analyzer -> Top():** Stack **->** Element  **Modifier -> Pop():** Stack -> Element y Stack  **Analyzer ->isEmpty() :** Stack -> Element |

| **TAD <Queue>** |
| --- |
| Queue={offer,poll,front} |
| Inv={ Comparator(a,b) = (Q={E1, E2,E3,E4…En} ∧ Q.poll=E1 ) ∧ True } |
| **Main Operations:**  **Builder -> CreateQueue : ->**Queue  **Analyzer -> front():** Queue x Element -> Element  **Modifier ->** Offer(Enqueue): Queue x Element -> Queue  **Modifier ->** Poll(Dequeue): Queue -> Element ∧ Queue  **Analyzer ->** IsEmpty(): Queue-> Boolean |

| **TAD <Priority Queue>** |
| --- |
| Priority Queue={size, comparator} |
| Inv: {comparator(a,b)= True} |
| **Main Operations:**  **Builder -> CreatePriorityQueue():** Element -> PriorityQueue  **Modifier -> Offer(Enqueue)():** Queue x Element x Integer -> Queue  **Analyzer-> Peek(Front)():** Queue -> Element  **Modifier -> Poll(Dequeue)():**  Queue -> Element  **Analyzer -> IsEmpty():**  Priority Queue -> Boolean  **Analyzer -> Size():** Priority Queue -> Integer |

| **TAD <Heap>** |
| --- |
| Heap ={size, comparator} |
| Inv: {heap[˪(i/2)˩] >= heap[i] } (assuming it is zero-indexed) |
| **Main Operations:**  **Builder -> CreateHeap():**  -> heap  **Modifier -> Heapeafy():** array -> heap  **Modifier -> Insert():** element, heap -> heap  **Modifier -> Extract():**  heap -> Element  **Analyzer -> IsHeap():**  array -> boolean  **Analyzer -> IsEmpty():**  heap -> Boolean  **Analyzer -> Size():** heap -> Integer |

**HashTable**

| **CreateQueue(size,table,hashFunction,keyEqualityFunction)**  “Creates a new hash table”  pre: True ∧ (size,table,hashFunction,keyEqualityFunction)  { post: TablaHash } |
| --- |

| **Insert(HashTable x key x Element)**  “places value in a key”  pre: HashTable != Nil ∧ key != Nill ∧ ∀k1( k1 ∈ HastTable(Keys) ∧ k1 != key ∧ Element !=Nil ∧ HashTable=={(keyN,ElementN)}  post( HashTable={(keyN,ElementN), (key,Elemento)}) |
| --- |

| **Remove(HashTable x key)**  “removes value from key”  pre: HashTable != Nil ∧ key != Nill ∧ ∃k1( k1 ∈ HastTable(Keys) ∧ k1 = key ∧ HashTable={(keyN,ElementN), (key,Element)}  {post: HashTable={(keyN,ElementN)}} |
| --- |

| **Search(HashTable x key)**  “gets value from key”  pre: HashTable != Nil ∧ key != Nill ∧ ∃k1( k1 ∈ HastTable(Keys) ∧ k1 = key ∧ HashTable={(keyN,ElementN), (key,Element)}  {post: Element } |
| --- |

| **Constains(HashTable, key)**  “finds if it has a key”  pre: HashTable != Nil ∧ key != Nill ∧ ∃k1( k1 ∈ HastTable(Keys) ∧ k1 = key ∧ HashTable={(keyN,ElementN), (key,Element)}  {post: True (if TasTable has this key)} |
| --- |

**Stack**

| **CreateStack(push, pop, Top):**  “Creates a new Stack”  pre: True ∧ (push,poll,size)  {post: Stack} |
| --- |

| **Push(Stack,Element)**  “adds a element to the Stack, in the first position”  pre: Element!=nil ∧ (Stack!=nil V Stack={En..E3,E3,E1})  {post=Stack={Element, En..E3,E3,E1}} |
| --- |

| **Top(Stack)**  “takes the first element of the Stack”  pre=Stack!=nil V Stack={En..E3,E3,E1}  {post: En} |
| --- |

| **Pop(Stack)**  “takes and remove the first element of the Stack”  pre=Stack!=nil V Stack={En…E3,E3,E1}  {post= En ∧ Stack={En-1,...E3,E3,E1}} |
| --- |

| **isEmpty(Stack)**  “If the Stack isEmpty”  pre=Stack!=nil  {post= boolean->if(Stack=Ø) } |
| --- |

**Queue**

| **CreateQueue(offer,front,poll)**  “Creates a new queue”  pre=True ∧ (offer,front,poll)  {post=Queue} |
| --- |

| **Front(Queue)**  **“**Take the first element of the Queue**”**  pre: (Queue != nill V Queue={E1, E2,E3,E4…En}) ∧ !Queue.isEmpty()  {post=E1} |
| --- |

| **Offer(Queue x Element)**  “Add the element in the Queue in the ultimate position”  pre: Queue!=nil ∧ Element!=nil ∧ Queue={E1, E2…En}  {post= Queue=(E1, E2…En,Element)} |
| --- |

| **Poll(Queue)**  **“**Take and remove the first element of the Queue**”**  pre: (Queue != nill V Queue={E1, E2,E3,E4…En}) ∧ !Q.isEmpty()  {post= E1 ∧ Queue={E2,E3,E4…En}} |
| --- |

| **IsEmpty(Queue)**  “If the QueueisEmpty”  pre: Queue !=nill  {post=boolean ->if(Queue=Ø)} |
| --- |

**Priority Queue**

| **CreatePriorityQueue(size):**  “Creates a new priority queque”  {pre: Size ∧ True }  {post: PriorityQueue} |
| --- |

**Heap**

| **CreateHeap(**size, comparator**)**  “creates a new heap”  pre: true  {post: heap ={size, comparator} |
| --- |

| **Heapify(array)**  “it receives an array and turns it into a heap”  pre: True  {post: array.isHeap() = true} |
| --- |

| **Insert(heap, element)**  “it receives an element and adds it to the heap maintaining the its property”  pre: heap.isHeap()  {post: heap.isHeap() = true} |
| --- |

| **Extract(heap)**  “return and eliminates the first element in the heap”  pre: heap.isHeap()  {post: heap.isHeap() = true ∧ heap.size = heap.size-1} |
| --- |

| **IsHeap(array)**  “return true if the array is a heap, false if it is not”  pre: True  {post: true if it is a heap, false if not} |
| --- |

| **IsEmpty(heap)**  “return true if the heap is empty, false if it is not”  pre: True  {post: true if it is empty, false if not} |
| --- |

| **Size(heap)**  “return the amount of elements in the heap”  pre: True  {post: heap.size} |
| --- |