|  |
| --- |
| **TAD<Hash Table>** |
| Hash table={size,table,hashFunction,keyEqualityFunction} |
| Inv= {∀∀ ( ∈ table(Keys) y k2 ∉ table(keys) y ≠ k2 y hashFuction()≠ y hashFuction() ≠ k2 y hashFuction() ≠ hashFuction() } |
| **Main operations**  **Builder →CreationHashTable(**size**)**: size →HashTable  **Modifier → Insert(**key , Element**):** HashTable x key x Element →HashTable  **Modifier →Remove(key):** HashTable x key x Element →HashTable  **Analyzer → Search():** HashTable x Key → Element  **Analyzer → Contains():** HashTable x key → boolean  **Builder → clone():** HashTable\_1→HashTable\_2 |

|  |
| --- |
| **TAD <Stack>** |
| Stack= {push,pop, peek} |
| Inv = {Comparator(a,b)=(S={E1,E2,E3…En} S.pop=En) ∧ True } |
| **Main operations**  **Builder → CreateStack(): →** Stack  **Modifier → Push(**Element**):** Stack x Element → Stack  **Analyzer → Top():** Stack **→** Element  **Modifier → Pop():** Stack → Element y Stack  **Analyzer →isEmpty() :** Stack → Element  **Analyzer →size() :** → Integer  **Builder → clone():** Stack1→Stack2 |

|  |
| --- |
| **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  **Analyzer →size() :** → n (size)  **Builder → clone():** Queue1→Queue2 |

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

|  |
| --- |
| **TAD <Heap>** |
| Heap ={size, comparator} |
| Inv: {heap[˪(i/2)˩] >= heap[i] ∧ parent>left ∧ parent>right } (assuming it is zero-indexed) |
| **Main Operations:**  **Builder → CreateHeap():**  → heap  **Modifier → Heapify():** array → heap  **Modifier → Insert():** element, heap → heap  **Modifier → Extract():**  heap → Element  **Analyzer → IsEmpty():**  heap → Boolean  **Builder → clone():** Heap1→Heap2  **Analyzer →size() :** → Integer (size) |

**HashTable**

|  |
| --- |
| **CreateHashTable(size)**  “Creates a new hashTable”  pre: size > 0  { post: HashTable } |

|  |
| --- |
| **Insert(key,Element)**  “places value in a key”  pre: HashTable ≠ Nil ∧ key ≠ Nil ∧ Inv=True}  post( HashTable={(,E1), (k2,E2)...(kn-1,En-1)), (key,Elemento)}) |

|  |
| --- |
| **Remove(key)**  “removes value from key”  pre: HashTable ≠ Nil ∧ key ≠ Nill ∧ Inv=true  {post: HashTable={(,E1), (k2,E2)...(kn-1,En-1))} |

|  |
| --- |
| **Search( key)**  “gets value from key”  pre: HashTable ≠ Nil ∧ key ≠ Nill ∧ inv=true}  {post: Element } |

|  |
| --- |
| **Constains( key)**  “finds if it has a key”  pre: HashTable ≠ Nil ∧ key ≠ Nill ∧ inv=true}  {post: True (if TasTable has this key)} |

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

|  |
| --- |
| **Clone(**HashTable1**)**  **“**Return a clone (an object identical to his “parent”, but has different object reference and a different memory direction)”  pre: HashTable1 ≠ Nil  {post: hashtable2} |

**Stack**

|  |
| --- |
| **CreateStack():**  “Creates a new Stack”  pre: True  {post: Stack} |

|  |
| --- |
| **Push(Element)**  “adds a element to the Stack, in the first position”  pre: Element≠nil ∧ Stack≠nil  {post=Stack={E1,E2,E3...En+1}} |

|  |
| --- |
| **Top()**  “takes the first element of the Stack”  pre=Stack≠nil  {post: En} |

|  |
| --- |
| **Pop()**  “takes and remove the first element of the Stack”  pre=Stack≠nil  {post= En ∧ Stack={E1,E2,E3….En-1}} |

|  |
| --- |
| **isEmpty()**  “If the Stack isEmpty”  pre=Stack≠nil  {post= True if(Stack=Ø) } |

|  |
| --- |
| **Size()**  “find and return the Stack size”  pre: Stack≠ Nil  {post: size} |

|  |
| --- |
| **Clone(**Stack1**)**  **“**Return a clone (an object identical to his “parent”, but has different object reference and a different memory direction)”  pre: Stack1 ≠ Nil  {post: Stack2} |

**Queue**

|  |
| --- |
| **CreateQueue()**  “Creates a new queue”  pre=True  {post=Queue} |

|  |
| --- |
| **Front(Queue)**  **“**Take the first element of the Queue**”**  pre: (Queue ≠ nill) ∧ !Queue.isEmpty()  {post=E1} |

|  |
| --- |
| **Offer(Element)**  “Add the element in the Queue in the last position”  pre: Queue≠nil ∧ Element≠nil ∧ Queue={E1, E2…En}  {post= Queue=(E1, E2…En-1,Element) } |

|  |
| --- |
| **Poll()**  **“**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()**  “If the QueueisEmpty”  pre: Queue ≠nill  {post=True if(Queue=Ø)} |

|  |
| --- |
| **Size()**  “find and return the Queue size”  pre: Queue≠ Nil  {post: size} |

|  |
| --- |
| **Clone(**Queue1**)**  **“**Return a clone (an object identical to his “parent”, but has different object reference and a different memory direction)”  pre: Queue1 ≠ Nil  {post: Queue2} |

**PriorityQueue**

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

|  |
| --- |
| **Offer(Element):**  “Adds Element to queque”  {pre: Element}  {pos: print PriorityQueque Elements with enquque Element } |

|  |
| --- |
| **Peek()**  “print queque ”  {pre: PriorityQueue≠ null}  {pos: prints queque} |

|  |
| --- |
| **Poll()**  “Removes Element from queque ”  {pre: PriorityQueue≠ null}  {pos: print PriorityQueque Elements with dequeque Element } |

|  |
| --- |
| **size():**  **Size()**  “find and return the PrioriyQueue size”  pre: PriorityQueue≠ null  {post: size} |

**Heap**

|  |
| --- |
| **CreateHeap(**size**)**  “creates a new heap”  pre: size>0  {post: heap |

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

|  |
| --- |
| **Insert(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} |

|  |
| --- |
| **Clone(**Heap1**)**  **“**Return a clone (an object identical to his “parent”, but has different object reference and a different memory direction)”  pre: Heap1 ≠ Nil  {post: Heap2} |