**TADs**

|  |  |  |
| --- | --- | --- |
| **HashTable** | | |
| **HashTable** = {Size = <size>, Elements= <Element\_1 = <key, value> ... Element\_n = <key, value>} | | |
| {**inv:** All key elements in the hashtable must be different} | | |
| **Primitive Operations:**  CreateHashTable  Put  Remove  Clear  ShowHashTable  Size  Search | HashTable X Key X Value  HashTable X Key  HashTable  HashTable  HashTable  HashTable X Key | → HashTable  → HashTable  → HashTable  → HashTable  → String  → Integer  → Element |

**Construction operations**

|  |
| --- |
| **CreateHashTable(Size)**  “Create a new empty hash table”  { **pre**: Size > 0 }  { **post**: HashTable = {HashTable = { Size = Size, Elements = {∅} }} } |

**Modifying operations**

|  |
| --- |
| **Put(HashTable, Key, Value)**  “Adds a key-value pair into the hash table.”  { **pre**: HashTable = { … } ⋀ Key = <key> ⋀ Value = <value>}  { **post**: HashTable = { Size = Size', Elements = Elements' ∪ {(Key, Value)} }} |

|  |
| --- |
| **Remove(Key, Value)**  “Remove the key-value pair associated with the specified key from the hash table.”  { **pre**: HashTable = { … } ⋀ (Key, Value) is in Elements }  { **post**: HashTable = { Size = Size', Elements = Elements' - {(Key, Value)} } } |

|  |
| --- |
| **Clear(HashTable)**  “Remove all key-value pairs from the hash table, effectively resetting it to an empty state.”  {**pre**: HashTable = { … } }  {**post**: HashTable = {HashTable = { Size = Size, Elements = {∅} }} } |

**Analysing operations**

|  |
| --- |
| **ShowHashTable(HashTable)**  “Return a string representation of the hash table.”.  {**pre**: HashTable = { … } }  {**post**: HashTable ∈ String} |

|  |
| --- |
| **Size(HashTable)**  “Return the number of key-value pairs in the hash table.”  {**pre**: HashTable = { …} }  {**post**: |Elements| } |

|  |
| --- |
| **Search( key)**  “Search for a key in the hash table and return the associated value ”  {**pre**: HashTable = { … } ⋀ Key ∈ HashTable}  {**post**: Value = <value>} |

|  |  |  |
| --- | --- | --- |
| **PriorityQueue** | | |
| **PriorityQueue** = { Elements = { Element\_1 = {PriorityCriterion =<priorityCriterion>, Value= <value>} … Element\_n = {PriorityCriterion =<priorityCriterion>, Value = <value>} } } | | |
| {**Inv:** The element at the top of the queue should be the item with the highest priority} | | |
| **Primitive Operations:**  CreatePriorityQueue  Add  Poll  IsEmpty  GetMax  ShowPriorityQueue | PriorityQueue X PriorityCriterion X Value  PriorityQueue  PriorityQueue  PriorityQueue  PriorityQueue | → PriorityQueue  → PriorityQueue  → Element  → Boolean  → Value  → String |

**Construction operations**

|  |
| --- |
| **CreatePriorityQueue()**  “Create a new empty priority queue”  { pre: TRUE }  { post: PriorityQueue = {∅} } |

**Modifying operations**

|  |
| --- |
| **Add(Element)**  “Add a new element in the PriorityQueue ”  { **pre:** PriorityQueue = { … } ⋀ Element= { … }  { **post:** Element ∈ PriorityQueue = { … } } |

|  |
| --- |
| **Poll()**  “Remove and returns the element at the top of the priority queue ”  { **pre:** PriorityQueue = { … } ⋀ PriorityQueue != {∅} }  { **post**: Element ∉ PriorityQueue ⋀ PriorityQueue = PriorityQueue' - {Element} ⋀ **returns** Element} |

**Analysing operations**

|  |
| --- |
| **isEmpty()**  “Check if the priority queue is empty”  { **pre:** PriorityQueue = { … } }  { **post**: BOOLEAN (PriorityQueue = {∅} } |

|  |
| --- |
| **GetMax()**  "Return the element with the highest priority without removing it from the priority queue."  { **pre:** PriorityQueue = { … } ⋀ PriorityQueue ≠ {∅} }  { **post:** Element ∈ PriorityQueue ⋀ ∀ n : Element| n ∈ PriorityQueue : n.PriorityCriterion ≤ Element.PriorityCriterion ⋀ **return** Element } |

|  |
| --- |
| **ShowPriorityQueue()**  "Return a string representation of the priority queue."  { **pre:** PriorityQueue = { … } }  { **post:** PriorityQueue = { … } ∈ String} |

|  |  |  |
| --- | --- | --- |
| **Queue** | | |
| **Queue** = { Elements = {Element\_1 … Element\_n }, Front = <front>, Size = <size> } | | |
| {**Inv:** The element at the front of the queue is the first one to be removed. FIFO (Fisrt In First Out) } | | |
| **Primitive Operations:**  CreateQueue  Enqueue  Dequeue  IsEmpty  Front  Size  ShowQueue | Queue X Element  Queue  Queue  Queue  Queue  Queue | → Queue  → Queue  → Element  → Boolean  → Element  → Integer  → String |

**Construction operations**

|  |
| --- |
| **CreateQueue()**  “Create a new empty queue”  { pre: TRUE }  { post: Queue = {∅} } |

**Modifying operations**

|  |
| --- |
| **Enqueue(Element)**  “Add a new element in the PriorityQueue ”  { **pre:** PriorityQueue = { … } ⋀ Element= { … }  { **post:** Element ∈ PriorityQueue = { … } } |

|  |
| --- |
| **Dequeue()**  “Remove and returns the element at the front of the queue”  { **pre:** Queue = { … } ⋀ Queue != {∅} }  { **post**: Element ∉ Queue ⋀ Queue = Queue' - {Element} ⋀ **returns** Element} |

**Analysing operations**

|  |
| --- |
| **IsEmpty()**  “Check if the queue is empty”  { **pre:** Queue = { … } }  { **post**: BOOLEAN (Queue = {∅} } |

|  |
| --- |
| **Front()**  "Return the element in the front of the queue without removing it from the queue."  { **pre:** Queue = { … } ⋀ Queue ≠ {∅} }  { **post:** Element ∈ Queue ⋀ ∀ n : Element| n ∈ Queue: n.Queue ≤ Element.Queue ⋀ **return** Element } |

|  |
| --- |
| **Size()**  "Return the number of elements in the queue”  { **pre:** Queue = { … } }  { **post**: |Elements| } |

|  |
| --- |
| **ShowQueue()**  "Return a string representation of the queue."  { **pre:** Queue = { … } }  { **post:** Queue = { … } ∈ String} |

|  |  |  |
| --- | --- | --- |
| **Stack** | | |
| **Stack =** { Elements = {Element\_1 … Element\_n }, Top = <top> } | | |
| {**Inv:** The last element to be added to the stack is the first one to be removed. LIFO (Last In First Out)} | | |
| **Primitive Operations:**  CreateStack  Push  Pop  Peek  isEmpty  ShowStack | Stack X Element  Stack  Stack  Stack  Stack | → Stack  → Stack  → Element  → Element  → Boolean  → String |

**Construction operations**

|  |
| --- |
| **CreateStack()**  “Create a new empty stack”  { **pre:** TRUE }  { **post**: Stack = {∅} } |

**Modifying operations**

|  |
| --- |
| **Push(Element)**  “Add a new element in the stack”  { **pre:** PriorityQueue = { … } ⋀ Element = { … }  { **post:** Element ∈ PriorityQueue = { … } ⋀ Top = Element} |

|  |
| --- |
| **Pop()**  “Remove and return the element at the top of the stack.”  { **pre:** Stack = { … } ⋀ Stack != {∅} }  { **post**: Element ∉ Stack ⋀ Stack = Stack - {Element} ⋀ Top = Element\_n-1 ⋀ **return** Element} |

**Analysing operations**

|  |
| --- |
| **Peek()**  "Return the element at the top of the stack without removing it”  { **pre:** Stack = { … } ⋀ Stack != {∅}}  { **post**: **return** top } |

|  |
| --- |
| **isEmpty()**  “Checks if the stack is empty”  { **pre:** stack = { … } }  { **post**: BOOLEAN (Stack = {∅} } |

|  |
| --- |
| **ShowStack()**  "Returns a string representation of the stack."  { **pre:** Stack = { … } }  { **post:** Stack = { … } ∈ String} |

|  |  |  |
| --- | --- | --- |
| **LinkedList** | | |
| **LinkedList** = { Head = <head>, Tail = <tail>, Elements = {Element\_1 = { Value = <value>, Next = <next>, Preve <prev>} … Element\_n = { Value = <value>, Next = <next>, Preve <prev>} } | | |
| {**Inv**: Each element in the linked list must have a valid value and a valid reference to the next element or must have a null reference if it is the last element in the list} | | |
| **Primitive Operations:**  CreateLinkedList  Add  Remove  Search  Size  isEmpty  ShowLinkedList | LinkedList X Element  LinkedList X Element  LinkedList X Value  LinkedList  LinkedList  LinkedList | → CreateLinkedList  → CreateLinkedList  → CreateLinkedList  → Element  → Integer  → Boolean  → String |

**Construction operations**

|  |
| --- |
| **CreateLinkedList()**  “Create a new empty linked list”  { **pre:** TRUE }  { **post**: LinkedList = {∅} } |

**Modifying operations**

|  |
| --- |
| **Add(Element)**  “Add a new element in the linked list”  { **pre:** LinkedList = { … } ⋀ Value = <value>, Next = <null>, Preve <prev> }  { **post:** Element ∈ LinkedList= { … } ⋀ LinkedList = { Head = Head', Tail = Element, Elements = Elements' ⋃ {Element} }} |

|  |
| --- |
| **Remove(Element)**  “Remove a specific element from the linked list.”  { **pre:** LinkedList = { … } ⋀ Element ∈ Elements }  { **post**: LinkedList = { Head = Head', Tail = Tail', Elements = Elements' - {Element} } } |

**Analysing operations**

|  |
| --- |
| **Search(Value)**  "Search for an element with a specific value in the linked list and return it if found”  { **pre:** LinkedList = { … } ⋀ Value ∈ LinkedList ⋀ LinkedList != {∅} }  { **post**: **return** Element } |

|  |
| --- |
| **Size()**  "Return the number of elements in the linked list”  { **pre:** LinkedList = { … } }  { **post**: |Elements| } |

|  |
| --- |
| **isEmpty()**  “Checks if the linked list is empty”  { **pre:** linkedList = { … } }  { **post**: BOOLEAN (LinkedList = { Head = null, Tail = null, Elements = {} }) } |

|  |
| --- |
| **ShowLinkedList()**  "Returns a string representation of the linked list."  { **pre:** LikedList= { … } }  { **post:** LinkedList= { … } ∈ String} |

|  |  |  |
| --- | --- | --- |
| **Heap** | | |
| **Heap** = { Capacity = <capacity>, Size = <size>,  Elements = { Element\_1 = { Value = <value>} … Element\_n = { Value = <value>} } } | | |
| {**Inv**: Each node in the linked list must have a valid value and a valid reference to the next node or must have a null reference if it is the last node in the list} | | |
| **Primitive Operations:**  Heap  Heapify  Insert  Remove  Peek  Size  ShowHeap | Heap  Heap X Node  Heap X Node  Heap  Heap  Heap | → Heap  → Heap  → Heap  → Heap  → Node  → Integer  → String |

**Construction operations**

|  |
| --- |
| **CreateHeap()**  “Create a new empty heap”  { **pre**: TRUE }  { **post**: heap = {∅} } |

**Modifying operations**

|  |
| --- |
| **Heapify(Elements)**  “Transform an array of elements into a valid heap”  { **pre:** Elements = { … }  { **post:** The array of elements is transformed into a valid heap} |

|  |
| --- |
| **Insert(Element)**  “Insert a new element into the heap while maintaining the heap property”  { **pre:** Heap = { … } ⋀ Element = { …} The element must be a valid element with a defined value}  { **post**: Element ∈ Heap = { … } ⋀ Heap = Heap' + {Element} } |

|  |
| --- |
| **Remove()**  “Remove and return the top (minimum) element from the heap while maintaining the heap property.”  { **pre:** Heap = { … } ⋀ Heap != {∅} }  { **post**: Element ∉ Heap ⋀ Heap = Heap’ - {Element} ⋀ **return** Element} |

**Analysing operations**

|  |
| --- |
| **Peek()**  "Get the top (minimum) element from the heap without removing it.”  { **pre:** Heap = { … } ⋀ Heap != {∅}}  { **post**: **return** top } |

|  |
| --- |
| **Size()**  "Get the current size (number of elements) of the heap.”  { **pre:** Heap = { … } }  { **post**: |Elements| } |

|  |
| --- |
| **ShowHeap()**  "Return a string representation of the heap."  { **pre:** Heap = { … } }  { **post:** Heap = { … } ∈ String} |