**HASHTABLE:**

| **Hashtable ADT** | | | |
| --- | --- | --- | --- |
| *{Elements = { (K1, V1), (K2, V2)} Elements = Ø}* | | | |
| *{ inv: 0 <= size <= TABLE\_SIZE }* | | | |
| **Operation** | **Type** | **Input** | **Output** |
| Hashtable | *Constructor* | - | Hashtable |
| put | *Modifier* | key , value | Hashtable |
| get | *Analyzer* | Key | Element |
| remove | *Modifier* | Key | Element |
| contains | *Analyzer* | Key | Boolean |
| isEmpty | *Analyzer* | - | Boolean |
| size | *Analyzer* | - | int |

| **Hashtable()**  “This constructor creates a new empty hash table.”  {pre: No specific precondition}  {pos: A new empty hash table is created with no key-value pairs.} |
| --- |
| **Put(K key, V value)**  {pre: No specific precondition}  {pos: The hash table contains a new key-value pair (key, value)} |
| **Get(K key)**  {pre: The key `key` is not null}  {pos: If the key `key` exists in the hash table, the method returns the associated value; otherwise, it returns `null`} |
| **Remove(K key)**  {pre: The key `key` is not null}  {pos: If an entry with key `key` exists, it is removed from the hash table; otherwise, the table remains unchanged} |
| **Contains(K key)**  {pre: The key `key` is not null}  {pos: It returns `true` if the hash table contains an entry with key `key`; otherwise, it returns `false`} |
| **IsEmpty()**  {pre: No specific precondition}  {pos: It returns `true` if the hash table is empty (contains no elements); otherwise, it returns `false`} |
| **Size()**  **{**pre: No specific precondition}  {pos: It returns the number of elements currently stored in the hash table} |

**QUEUE:**

| **Queue ADT** | | | |
| --- | --- | --- | --- |
| { *Elements = {, ,,…, } Elements = Ø,*  Front, Back} | | | |
| { } | | | |
| **Operation** | **Type** | **Input** | **Output** |
| Queue | *Constructor* | - | Queue |
| enqueue | *Modifier* | Queue x Element | Queue |
| dequeue | *Modifier* | Queue | Element |
| front | *Analyzer* | Queue | Element |
| isEmpty | *Analyzer* | Queue | Boolean |

| **Queue()**  “Builds an empty queue “  {pre: TRUE}  {pos: Queue q = {Elements = Ø, front = Ø, Back = Ø}} |
| --- |
| **Enqueue(q, e)**  “Inserts a new element e to the back of the queue q”  {pre: q = {Elements = Ø, front = Ø, back = Ø} q = {Elements = {, ,,…, }, front = , back = } Element e}  {pos: q = {Elements = e, front = e, back = e} s = {Elements = {, ,,…, , e}, front = , back = }} |
| **Dequeue(q)**  “Extracts the element in Queue q’s front”  {pre: q {Elements = Ø, front = Ø, back = Ø} i.e q = {Elements = {, ,,…, }, front =  , back = }  {pos: {pos: q = {Elements = {, ,,…, }, front = , back = } e = } |
| **Front(q)**  “Recovers the value of the element on the front of the queue”  {pre: q {Elements = Ø, front = Ø, back = Ø} i.e q = {Elements = {, ,,…, }, front =  , back = }  {pos: Element } |
| **IsEmpty(q)**  “Determines if the Queue q is empty or not”  {pre: q = {Elements = Ø, front = Ø, back = Ø} q = {Elements = {, ,,…, }, front = , back = }}  {pos: True if q = {Elements = Ø, front = Ø, back = Ø} False if q{Elements = Ø, front = Ø, back = Ø}} |

**STACK:**

| **Stack ADT** | | | |
| --- | --- | --- | --- |
| Stack = { *Elements = {, ,,…, } Elements = Ø,*  Top} | | | |
| { } | | | |
| **Operation** | **Type** | **Input** | **Output** |
| Stack | *Constructor* | - | Stack |
| push | *Modifier* | Stack x Element | Stack |
| pop | *Modifier* | Stack | Element |
| top | *Analyzer* | Stack | Element |
| isEmpty | *Analyzer* | Stack | Boolean |

| **Stack()**  “Creates a new Stack with empty data.”  {pre: TRUE}  {post: Stack s = {Elements = Ø, Top = Ø}} |
| --- |
| **Push(s, e)**  “Adds the new Element e to stack s”  {pre: s = {Elements = Ø, Top = Ø} s = {Elements = {, ,,…, }, Top = } Element e}  {post: s = {Elements = e, Top = e}s = {Elements = {, ,,…, , e}, Top = e}} |
| **Pop(s)**  “Returns and extract from the stack s, the most recently inserted element”  {pre: s {Elements = Ø, Top = Ø} i.e s = {Elements = {, ,,…, }, Top = }  {pos: s = {Elements = {, ,,…, }, Top = } e = } |
| **Top(s)**  “Recovers the value of the element on the top of the stack”  {pre: s {Elements = Ø, Top = Ø} i.e s = {Elements = {, ,,…, }, Top = }  {post: e = } |
| **IsEmpty(s)**  “Determines if the stack s is empty or not”  {pre: s = {Elements = Ø, Top = Ø}s = {Elements = {, ,,…, }, Top = }  {pos: True if s s = {Elements = Ø, Top = Ø} False if s {Elements = Ø, Top = Ø}} |

**PriorityQueue:**

| **PriorityQueue ADT** | | | |
| --- | --- | --- | --- |
| PriorityQueue = { Elements = {e1, e2, e3, ..., en}, Priority = {p1, p2, p3, ..., pn}, Elements = Ø, Front} | | | |
| { inv: (Size(PriorityQueue) = n AND Front = e1), Elements = {e1, e2, e3, ..., en}, Priority = {p1, p2, p3, ..., pn}, Elements = Ø, Front} | | | |
| **Operation** | **Type** | **Input** | **Output** |
| PriorityQueue | *Constructor* | - | PriorityQueue |
| getRoot | *Analyzer* | PriorityQueue | element |
| *insert* | *Modifier* | PriorityQueue X elemenT |  |
| extractRoot | *modifier* | PriorityQueue | Element |
| setKey | *modifier* | PriorityQueue X int X element |  |
| resize | *modifier* | PriorityQueue X int |  |
| isFull | *analyzer* | PriorityQueue | Boolean |

| **PriorityQueue()**  "Creates a new PriorityQueue with empty data."  {pre: TRUE}  {post: PriorityQueue pq = {Elements = Ø, Front = Ø} |
| --- |
| **getRoot(pq)**  “Gets the root element of the PriorityQueue pq”  {pre: pq = {Elements = Ø, Front = Ø} OR pq = {Elements = {e1, e2,e3,…, en }, Front = e1}}  {post: pq = {Elements = Ø, Front = Ø} OR pq = {Elements = {e1, e2,e3,…, en }, Front = e1} AND Element e1} |
| **insert(pq, e)**  “Inserts the new Element e into PriorityQueue pq”  {pre: pq = {Elements = Ø, Front = Ø} OR pq = {Elements = {e1, e2,e3,…, en }, Front = e1} AND Element e}  {post: pq = {Elements = e, Front = e} OR pq = {Elements = {e1, e2,e3,…, en , e}, Front = e1}} |
| **extractRoot(pq)**  “Extracts the root element from PriorityQueue pq”  {pre: pq = {Elements = {e1, e2,e3,…, en }, Front = e1}}  {post: pq = {Elements = {e2,e3,…, en }, Front = e2} AND Element e1} |
| **setKey(pq, i, e)**  “Sets the Element e at index i in PriorityQueue pq”  {pre: pq = {Elements = {e1, e2,e3,…, en }, Front = e1} AND int i AND Element e}  {post: pq = {Elements = {e1, e2,e3,…, ei=e ,…, en }, Front = e1}} |
| **resize(pq, capacity)**  “Resizes the PriorityQueue pq to the new capacity”  {pre: pq = {Elements = {e1, e2,e3,…, en }, Front = e1} AND int capacity}  {post: pq = {Elements = {e1, e2,e3,…, en }, Front = e1} AND Size(pq) = capacity} |
| **isFull(pq)**  “Checks if the PriorityQueue pq is full”  {pre: pq = {Elements = {e1, e2,e3,…, en }, Front = e1}}  {post: pq = {Elements = {e1, e2,e3,…, en }, Front = e1} AND Boolean} |

| **Heap ADT** | | | |
| --- | --- | --- | --- |
| Heap = { Elements = {e1, e2, e3, …, en}, Elements = Ø} | | | |
| { inv: (Size(Heap) = n AND heap[0] = e1), Elements = {e1, e2, e3, …, en}, Elements = Ø} | | | |
| **Operation** | **Type** | **Input** | **Output** |
| Heap | *Constructor* | - | Heap |
| invariant | *Analyzer* | Heap X int | Boolean |
| parent | Analyzer | Heap x int | int |
| left | Analyzer | Heap x int | int |
| right | Analyzer | Heap x int | int |
| swap | *modifier* | Heap x int x int | - |
| heapify | Modifier | Heap x int | - |
| buildHeap | Modifier | Heap | - |
| toString | Analyzer | Heap | String |
| heapSort | Modifier/Analyzer | Heap | Heap |

| **Heap()**  **“Initializes a new instance of the Heap class” {pre: -} {post: h = {Elements = Ø}, size = 0}** |
| --- |
| **invariant(h, i)**  Checks if the heap maintains its invariant at position i” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}, i is a valid index in the heap} {post: Returns true if the element at position i is greater than or equal to its children (if any), otherwise returns false} |
| **parent(i)**  “Returns the index of the parent of the node at position i” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}, i is a valid index in the heap} {post: Returns the index of the parent of the node at position i} |
| **left(i)**  **“**Returns the index of the left child of the node at position i” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}, i is a valid index in the heap} {post: Returns the index of the left child of the node at position i} |
| **right(i)**  “Returns the index of the right child of the node at position i” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}, i is a valid index in the heap} {post: Returns the index of the right child of the node at position i} |
| **swap(i, j)**  “Swaps two nodes at positions i and j” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}, i and j are valid indices in the heap} {post: The elements at positions i and j in the heap have been swapped} |
| **heapify(i)**  Reorganizes the heap to maintain its invariant starting from position i” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}, i is a valid index in the heap} {post: The heap maintains its invariant starting from position i} |
| **buildHeap()**  “Builds a heap from an array” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}} {post: The array has been reorganized into a heap} |
| **toString()**  “Returns a string representation of the heap” {pre: h = {Elements = Ø} OR h = {Elements = {e1, e2, e3,…, en}}} {post: Returns a string representation of the heap |