**TADs**

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table TAD** | | | |
| { inv: keyi = keyk ↔ i = j } | | | |
| Primitive operations: | | | |
| * HashTable | Integer | * HashTable | Constructor |
| * hashFunction | Key | * Integer | Analyzer |
| * getAll | HashTable | * Array | Analyzer |
| * Insert | Value | * String | Constructor |
| * searchNode | HashTable X Key | * Value | Analyzer |
| * set | Value | * Value | Modifier |
| * delete | HashTable X Key | * HashTable | Modifier |
| * clone | HashTable | * HashTable | Constructor |

|  |
| --- |
| **HashTable(***s***)** |
| Constructor for the HashTable class. Initializes a hash table with the specified size.  { pre: TRUE}  { post: *HashTable* = { array = NIL , size = s } |

|  |
| --- |
| **hashFunction(***key***)** |
| Computes the hash code for a given key, ensuring it falls within the array's size boundaries.  { pre: TRUE }  { post: n ≥ 0, n < size} |

|  |
| --- |
| **insert(***hashTable, key, value***)** |
| Adds a key-value pair to the hash table using separate chaining for collision resolution.  { pre: TRUE }  { post: array = <key1 → value1 >, <…> , <key→ value>, <…> , < keyn → valuen > } |

|  |
| --- |
| **search(***hashTable, key***)** |
| Retrieves the value associated with a given key from the hash table.  { pre: TRUE }  { post: <keyi = key , valuei > } |

|  |
| --- |
| **getAll***(hash}Table)* |
| Retrieves all the values saved in the hash table  { pre: array = { <key1 → value1 >, <…>, <keyi = key, valuei >, <…> , = <keyn → valuen > } }  { post: array = { <key1 → value1 >, <…> , = <keyn → valuen > } } |

|  |
| --- |
| **set(***hashTable, key, value***)** |
| Updates the value associated with a given key in the hash table.  { pre: <keyi = key , valuei > }  { post: <keyi , valuei = value> } |

|  |
| --- |
| **searchNode(***hashTable, key***)** |
| Checks if the hash table contains a specific node.  { pre: array = <key1 → value1 >, <…> , < keyk → valuek >, <…> , < keyn → valuen > }  { post: True if <key1 → value1 >, <…> , < keyk = key→ valuek >, <…> , < keyn → valuen > } |

|  |
| --- |
| **delete(***key***)** |
| Removes a key-value pair from the hash table based on the key.  { pre: array = { <key1 → value1 >, <…>, <keyi = key, valuei >, <…> , = <keyn → valuen > } }  { post: array = { <key1 → value1 >, <…> , = <keyn → valuen > } } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Stack TAD** | | | |
| Stack = { <E1>, <E2>, <…> , <En> } | | | |
| { inv: TRUE } | | | |
| Primitive operations: | | | |
| * Stack |  | * Stack | Constructor |
| * push | Stack X Element | * Stack | Modifier |
| * pop | Stack | * Element | Modifier |
| * isEmpty | Stack | * Boolean | Analyzer |
| * size | Stack | * Integer | Analyzer |

|  |
| --- |
| **Stack** |
| Constructor for the Stack class. Initializes a new stack with a default capacity.  { pre: TRUE}  { post: stack = { data = NIL, size = 0 } } |

|  |
| --- |
| **push(***stack, elem***)** |
| Pushes an element on the top of the stack.  { pre: TRUE }  { post: queue = { <E1>, <…> , <En = elem> } } |

|  |
| --- |
| **pop(***queue***)** |
| Removes and returns the element from the top of the stack.  { pre: size > 0 }  { post: pop = <En>, queue = { <E1>, <…> , <En> } } |

|  |
| --- |
| **isEmpty(***queue***)** |
| Checks if the stack is empty.  { pre: TRUE }  { post: True if queue = { data = NIL, size = 0 }  False if queue = { data != NIL, size > 0 } } |

|  |
| --- |
| **size(***queue***)** |
| Returns the number of elements in the stack.  { pre: TRUE }  { post: size } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Queue TAD** | | | |
| Queue = { <E1>, <E2>, <…> , <En>} | | | |
| { inv: TRUE } | | | |
| Primitive operations: | | | |
| * Queue |  | * Queue | Constructor |
| * isEmpty | Queue | * Boolean | Analyzer |
| * front | Queue | * Element | Analyzer |
| * dequeue | Queue | * Element | Modifier |
| * enqueue | Queue X Element | * Queue | Modifier |
| * size | Queue | * Integer | Analyzer |

|  |
| --- |
| **Queue** |
| Constructor of the Queue class that creates an empty queue. { pre: TRUE}  { post: queue = { front = NIL, back = NIL, n = 0 } } |

|  |
| --- |
| **isEmpty(***queue***)** |
| Determines whether the queue is empty. Returns 'true' if the queue is empty and 'false' if it contains elements.  { pre: TRUE }  { post: True if queue = { front = NIL, back = NIL, n = 0 }  False if queue = { front != NIL, n > 0 } } |

|  |
| --- |
| **front(***queue***)** |
| Returns the element at the front of the queue without removing it.  { pre: n > 0 }  { post: peek = <E1> } |

|  |
| --- |
| **dequeue(***queue***)** |
| Remove and return the element at the beginning of the queue.  { pre: n > 0 }  { post: poll = <E1>, queue = { <E2>, <…> , <En> } } |

|  |
| --- |
| **enqueue(***queue, elem***)** |
| Adds an element to the end of the queue.  { pre: TRUE }  { post: queue = { <E1>, <E2>, <…> , <En = elem> } } |

|  |
| --- |
| **size(***queue***)** |
| Returns the number of elements in the queue.  { pre: TRUE }  { post: size = n } |

|  |
| --- |
| TAD <PriorityQueue> |
| PriorityQueue = sorts a list of objects, always getting out the one with the highest value. |
| {Inv: the object with the highest value (heap) will always be first at the priority queue.} |
| Operation:                           input:                               output:     |  |  |  | | --- | --- | --- | | insert | element | Inserts the element onto the priority queue, making the heap grow bigger. | | Extract-Max |  | Gets the element with the highest value and eliminates it, this is the method that marked the priority queue advance. | | Increase-Key | newKey | Compared with the older key, if the new isn’t smaller it will increase the value, making it higher on the priority queue. | | Maximum |  | |  | | --- | |  |   Returns the element with the highest value.   |  | | --- | |  | | |