Definición de los TAD de las estructuras a utilizar

|  |
| --- |
| TAD Hash Table |
| Una tabla hash es una estructura de datos que almacena valores en un arreglo en forma de diccionario. Es decir, cuando se ingresa un objeto a la tabla hash, se hace mediante una llave y un valor, la llave se utiliza para determinar la posición que el objeto tendrá en el arreglo mediante una función hash. Dicha función hash es de la forma donde k es la clave y m un número entero. Es decir que la función hash se encarga de transformar las claves que recibe en números enteros, esto con el objetivo de darle a los objetos una posición en el arreglo. Lo que se guardará en dicha posición será el valor del objeto. |
| El invariante de una tabla hash es que para una llave k, siempre se obtiene el mismo valor del código hash, independientemente de cuántas veces se realice la operación. Para todo k se cumple que cuando la ingreso a la función hash obtengo siempre la misma posición.  Teniendo k una clave, h la función hash, i una posición en el arreglo y m el tamaño del arreglo: |
| Operaciones primitivas:   * HashTable: size 🡪 HashTable * hashFunction: key 🡪 int * add: key x value 🡪 boolean * search: key 🡪 value * delete: key 🡪 value * getSize: 🡪 int |

|  |
| --- |
| **HashTable(size)**  Crea una nueva tabla hash vacía con el tamaño indicado por parámetro.  {pre: }  {post: HashTable HashTable.size = size} |

|  |
| --- |
| **hashFunction (key)**  Retorna una posición en el arreglo.  {pre: HashTable.}  {post: } |

|  |
| --- |
| **add (key, value)**  con la clave ingresada por parámetro asigna una posición en el arreglo al valor que también se recibe por parámetro. La posición es determinada por la función hash.  {pre: HashTable.}  {post: } |

|  |
| --- |
| **search (key)**  con la clave ingresada por parámetro retorna el valor que se encuentra en la posición que dicha clave mapea en el arreglo.  {pre: HashTable. }  {post: } |

|  |
| --- |
| **delete (key)**  con la clave ingresada por parámetro retorna el valor que se encuentra en la posición que dicha clave mapea en el arreglo y elimina dicho valor del arreglo.  {pre: HashTable. }  {post: ,  } |

|  |
| --- |
| **getSize ()**  Retorna el tamaño del arreglo. No es necesariamente el número de elementos en la tabla, sino el espacio que se asigna en memoria para el arreglo.  {pre: HashTable. }  {post: size} |

|  |
| --- |
| TAD Priority Queue |
| Una cola de prioridad (Priority Queue) es una estructura de datos que maneja una secuencia de un número arbitrario de elementos. Es similar a una cola, pero los elementos tienen, adicionalmente, una prioridad asignada. Siendo x un nivel de prioridad:  PriorityQueue = { Ø | e } |
| El invariante de una cola de prioridad es que un elemento con mayor prioridad será desencolado antes que un elemento de menor prioridad y Si dos elementos tienen la misma prioridad, se desencolarán siguiendo el orden de cola.  {inv: { será eliminado primero que siempre que **∨** serán eliminados por orden de llegada }} |
| Operaciones primitivas:   * insert: item 🡪 void * peek: 🡪 item * poll: 🡪 item * isEmpty: 🡪 boolean |

|  |
| --- |
| **insert (item)**  inserta el elemento en la cola de prioridad.  {pre: TRUE }  {post: inserta el elemento en la cola de prioridad } |

|  |
| --- |
| **peek ()**  Retorna mas no remueve el elemento a la cabeza, es decir, con mayor prioridad.  {pre: TRUE }  {post: la cabeza de la cola, null si esta vacia } |

|  |
| --- |
| **poll ()**  Retorna Y remueve el elemento a la cabeza, es decir, con mayor prioridad.  {pre: TRUE }  {post: la cabeza de la cola, null si esta vacia priorityqueue.size = priorityqueue.size -1} |

|  |
| --- |
| **isEmpty ()**  indica si la priority queue esta vacia .  {pre: TRUE}  {post: TRUE si la priority queue esta vacia , FALSE de lo contrario} |

|  |
| --- |
| TAD Max-Heap |
| El max-heap o montículo máximo es un arreglo que puede verse como un árbol binario casi completo, en donde cada nodo del árbol corresponde a un elemento del arreglo. Siendo x un nivel de prioridad:  Max-Heap = { Ø | e } |
| El invariante de un montículo máximo es que la longitud de toda rama es h o h – 1, siendo h la altura del árbol, además, no puede existir una rama de longitud h a la derecha de una rama de longitud h − 1. Además, en particular para el max-heap, Para todos los nodos i, excluyendo la raíz, A[Padre(i)] ≥ A[i]. |
| Operaciones primitivas:   * Max-heapify: arreglo x indice 🡪 void * build-max-heap: arreglo 🡪 max-heap * parent: indice 🡪 indice * leftChild: indice 🡪 indice * rightChild: indice 🡪 indice * swap indice1, indice2 🡪 void * insert item 🡪 void * extractMax 🡪 item * heapMaximum 🡪 item * heapIncreasekey arreglo x indice x valor 🡪 void |

|  |
| --- |
| **Max-heapify (A, i)**  Se usa para garantizar la propiedad de orden del max-heap.  {pre: los subárboles izquierdos y derechos son max-heaps}  {post: el subárbol con raíz i es un max-heap } |

|  |
| --- |
| **build-max-heap (A)**  Convierte un arreglo que se recibe como parámetro en un max-heap.  {pre: A.length > 1 }  {post: el arreglo A es ahora un max-heap } |

|  |
| --- |
| **parent (i)**  Retorna el indice del padre del elemento que se recibe como parametro.  {pre: heap.size > 1 }  {post: el indice del padre del elemento que se recibe como parametro} |

|  |
| --- |
| **leftChild (i)**  Retorna el índice del hijo izquierdo del elemento que se recibe como parámetro.  {pre: heap.size > 1 }  {post: el índice del hijo izquierdo del elemento que se recibe como parámetro} |

|  |
| --- |
| **rightChild (i)**  Retorna el índice del hijo derecho del elemento que se recibe como parámetro.  {pre: heap.size > 1 }  {post: el índice del hijo derecho del elemento que se recibe como parámetro} |

|  |
| --- |
| **swap (i, j)**  Intercambia las posiciones de los elementos que se encuentran en los índices recibidos como parámetro.  {pre: heap.size > 1 ≠}  {post: Intercambia las posiciones de los elementos que se encuentran en los índices recibidos como parámetro } |

|  |
| --- |
| **insert (item)**  Agrega el elemento al heap.  {pre: TRUE}  {post: Se inserta en elemento en el heap. El heap cumple la propiedad de orden gracias a la función increaseKey } |

|  |
| --- |
| **extractMax ()**  Remueve el elemento de mayor prioridad en el heap.  {pre: : heap.size > 1 }  {post: Remueve el elemento de mayor prioridad en el heap heap.size = heap.size - 1 el heap cumple con la propiedad de orden gracias a la función maxHeapify accionada en la raíz después de haber puesto en la raíz al último elemento del heap.} |

|  |
| --- |
| **heapMaximum ()**  Retorna mas no remueve el elemento de mayor prioridad en el heap.  {pre: : heap.size > 1 }  {post: Retorna el elemento de mayor prioridad en el heap heap.size queda intacto } |

|  |
| --- |
| **heapIncreasekey (A, i, k)**  Esta función reemplaza el valor que se encuentre en la posición i del arreglo con el valor k que se recibe por parámetro, siempre que el valor sea mayor o igual al actual. Para garantizar de mantener la propiedad de orden, va “subiendo” por el árbol, es decir, evalúa todos los nodos padre a partir de la posición i para intercambiar con sus hijos aquellos padres que sean menores.  {pre: heap.size > 1 k }  {post: Retorna el elemento de mayor prioridad en el heap heap.size queda intacto } |

|  |
| --- |
| TAD Stack |
| Una pila (stack) es una secuencia lineal de un número arbitrario de elementos. Las operaciones que permite son la inserción y eliminación solamente al tope de la misma. El tope es el último elemento agregado. La pila es una lista que sigue un patron LIFO (el último en llegar es el primero en salir). Una pila puede estar vacía o tener una secuencia de elementos.  Stack ={ Ø } |
| El invariante de una pila es que el acceso a los elementos de la pila está limitado a hacerse siempre desde el tope.  {inv: { eliminar: 🡪 n , | n es el último elemento en haberse agregado insertar: n | n se inserta al tope de la pila }} |
| Operaciones primitivas:   * Stack: 🡪 Stack * isEmpty: 🡪 boolean * push: item 🡪 void * top: 🡪 item * pop: 🡪 item |

|  |
| --- |
| **Stack ()**  Crea una pila vacía.  {pre: TRUE }  {post: Stack = { top = null, isEmpty = TRUE }} |

|  |
| --- |
| **isEmpty ()**  Indica si la pila está vacía.  {pre: TRUE }  {post: TRUE si la pila está vacía, FALSE en caso contrario} |

|  |
| --- |
| **push (item)**  Agrega el elemento a la pila.  {pre: TRUE }  {post: item es el nuevo tope de la pila} |

|  |
| --- |
| **top (item)**  Retorna mas no remueve el elemento al tope de la pila.  {pre: stack.isEmpty() debe ser FALSE }  {post: item, el tope de la pila} |

|  |
| --- |
| **pop ()**  Retorna y remueve el elemento al tope de la pila.  {pre: stack.isEmpty() debe ser FALSE }  {post: item, el tope de la pila stack} |

|  |
| --- |
| TAD Queue |
| Una cola (queue) es una secuencia lineal de un número arbitrario de elementos. Las operaciones que permite son la inserción solamente al final de la cola y la eliminación solamente al frente de la misma. La cola es una lista que sigue un patron FIFO (el primero en llegar es el primero en salir). Una cola puede estar vacía o tener una secuencia de elementos.  Queue ={ Ø } |
| El invariante de una cola es que la inserción de elementos se puede hacer únicamente al final (back) de la cola y la eliminación de elementos se puede hacer solamente al frente (front) de la misma.  {inv: { eliminar: 🡪 n , | n es el elemento que se ha agregado de manera menos reciente insertar: n | n es el elemento que se ha agregado de manera más reciente y se inserta al final de la cola. }} |
| Operaciones primitivas:   * Queue: 🡪 Queue * isEmpty: 🡪 boolean * enqueue: item 🡪 void * front: 🡪 item * dequeue: 🡪 item |

|  |
| --- |
| **Queue ()**  Crea una cola vacía.  {pre: TRUE }  {post: **Queue** = { front = null, back = null, isEmpty = TRUE }} |

|  |
| --- |
| **isEmpty ()**  Indica si la cola está vacía.  {pre: TRUE }  {post: TRUE si la cola está vacía, FALSE en caso contrario} |

|  |
| --- |
| **enqueue (item)**  Agrega el elemento al final de la cola.  {pre: TRUE }  {post: item es el nuevo final de la cola} |

|  |
| --- |
| **front (item)**  Retorna mas no remueve el elemento al frente de la cola.  {pre: queue.isEmpty() debe ser FALSE }  {post: item, el frente de la cola} |

|  |
| --- |
| **dequeue ()**  Retorna y remueve el elemento al frente de la cola.  {pre: queue.isEmpty() debe ser FALSE }  {post: item, el frente de la cola stack} |