|  |
| --- |
| TAD<Stack> |
| |  | | --- | | Stack={ index <T>} | |
| { inv: the size of table should have a positive value } |
| peek()  push(T e)  <T> pop()  createStack(cosntructor) |

|  |
| --- |
| peek()  {devuelve el nodo líder del árbol, que es la raíz del árbol. Este método no elimina el nodo líder del árbol, simplemente lo devuelve.}  Pre:{ el árbol no debe estar vacío. Si el árbol está vacío, el método simplemente imprime una cadena vacía en la consola y devuelve nulo.}  Post:{ devuelve el nodo líder del árbol. Si el árbol no está vacío, el método devuelve el nodo líder. Si el árbol está vacío, el método devuelve nulo y muestra un mensaje vacío en la consola.} |

|  |
| --- |
| push(T key)  {agrega un nuevo nodo al inicio de la lista doblemente enlazada que representa al árbol.}  Pre:{ el parámetro "key" debe ser diferente de nulo}  Post:{ el método agrega un nuevo nodo al inicio de la lista doblemente enlazada que representa al árbol. Si el árbol está vacío, crea un nuevo nodo con el valor proporcionado} |

|  |
| --- |
| <T> pop()  {elimina el primer nodo de la lista doblemente enlazada que representa al árbol y devuelve ese nodo.}  Pre:{ el árbol no debe estar vacío. Si el árbol está vacío, el método muestra un mensaje en la consola y devuelve nulo.}  Post:{ elimina el primer nodo de la lista doblemente enlazada que representa al árbol y devuelve ese nodo. Si el árbol no está vacío, se establece el siguiente nodo como el nuevo líder del árbol y se actualizan los enlaces entre los nodos.} |

|  |
| --- |
| createStack(cosntructor)  {constructor de Stack}  Pre:{}  Post:{constructor} |

|  |
| --- |
| TAD<HEAP> |
| |  | | --- | | HEAP={ index <T>} | |
| { inv: the size of table should have a positive value } |
| add(T)  <T>poll()  siftDown(int index)  siftUP(int index)  swap( index1, index2 )  createHEAP(cosntructor) |

|  |
| --- |
| add(T)  {El elemento T queda adicionado al parámetro}  Pre: { el arreglo heapArray debe tener espacio disponible para agregar un nuevo elemento}  Post:{el elemento "value" se agregará al arreglo heapArray en la posición heapSize y la variable heapSize se incrementará en uno} |

|  |
| --- |
| <T>poll()  { Este método "poll()" elimina y devuelve el elemento con el valor más alto (la raíz) del heap.}  Pre:{ el heap no esté vacío, lo que se verifica al comprobar que la variable heapSize no sea igual a cero. Si heapSize es cero, significa que el heap está vacío y el método devuelve "null"}  Post:{ el elemento con el valor más alto (la raíz) se eliminará del arreglo heapArray y se devolverá el valor de la raíz eliminada.} |

|  |
| --- |
| siftDown(int index)  {se utiliza para mantener la propiedad de heap en el arreglo heapArray después de eliminar un elemento.}  Pre:{ el índice proporcionado debe ser válido para el arreglo heapArray.}  Post:{ para cualquier nodo en el heap, el valor de ese nodo debe ser mayor o igual que los valores de sus nodos hijos} |

|  |
| --- |
| siftUP(int index)  { se utiliza para mantener la propiedad de heap en el arreglo heapArray después de insertar un nuevo elemento.}  Pre:{ el índice proporcionado debe ser válido para el arreglo heapArray.}  Post:{ para cualquier nodo en el heap, el valor de ese nodo debe ser mayor o igual que los valores de sus nodos hijos} |

|  |
| --- |
| swap( index1, index2 )  {intercambiar los valores de dos elementos en el arreglo heapArray.}  Pre:{ los índices proporcionados deben ser válidos para el arreglo heapArray.}  Post:{ los valores en los índices proporcionados se intercambiarán en el arreglo heapArray.} |

|  |
| --- |
| createHeap(cosntructor)  {crea el heap}  Pre:{null}  Post:{ constructor} |

|  |
| --- |
| TAD<QUEUE> |
| |  | | --- | | QUEUE={ index <T>} | |
| { inv: the size of table should have a positive value } |
| Enqueue(T)  <T>dequeue()  <T>getFIrsteInQueue()  createQueue(constructor) |

|  |
| --- |
| Enqueue(T)  { agrega un elemento a una cola doblemente enlazada.}  Pre{ la cola doblemente enlazada existe y es válida}  Post{ el elemento se agrega correctamente al final de la cola} |

|  |
| --- |
| <T>dequeue()  { elimina y devuelve el primer elemento de una cola doblemente enlazada.}  Pre{ la cola no está vacía.}  Post{ se ha eliminado el primer elemento de la cola y se ha devuelto su valor, o bien se ha devuelto nulo si la cola estaba vacía. Además, el tamaño de la cola se ha reducido en uno.} |

|  |
| --- |
| <T> getFirstInQueue()  { devuelve el primer nodo en la cola doblemente enlazada, pero no lo elimina de la cola.}  Pre{ la cola doblemente enlazada debe tener al menos un elemento.}  Post{ devuelve el primer nodo en la cola doblemente enlazada sin eliminarlo de la cola. Si la cola está vacía, devuelve null.} |

|  |
| --- |
| createQueue(constructor)  {Constructor del metodo Queue}  Pre{null}  Post[Queue} |

|  |
| --- |
| TAD<HASHTABLE> |
| |  | | --- | | HASHTABLE={ index <T,K>} | |
| { inv: the size of table should have a positive value } |
| Hash(K)  Insert(K,T)  <T>search(K)  printTable()  createHash(Constructor) |

|  |
| --- |
| Hash(K)  { toma una cadena de caracteres k como entrada y devuelve un valor entero que representa el índice donde se almacenará la clave en una tabla hash}  Pre{ El parámetro k debe ser una cadena de caracteres.}  Post{ El método devuelve un valor entero que representa un índice en una tabla hash.} |

|  |
| --- |
| Insert(K,T)  { Este método inserta un nuevo elemento en una tabla hash\*}  Pre{ Debe existir una tabla hash donde insertar los elementos.}  Post{ l elemento y su clave se han insertado en la tabla hash y la estructura de datos ha sido actualizada en consecuencia.} |

|  |
| --- |
| <T>search(K)  { Este método busca un elemento en la tabla hash utilizando una clave K y un valor hash precalculado hash.}  Pre{ la tabla hash debe existir y no estar vacía.}  Post{ devuelve el valor asociado a la clave key si existe en la tabla hash, o null si no se encuentra.} |

|  |
| --- |
| printTable()  { El método recorre la tabla hash e imprime los elementos contenidos en ella.}  Pre{ La tabla hash debe existir y contener elementos.}  Post{ Devuelve una cadena de caracteres que representa la tabla hash.} |

|  |
| --- |
| createHash(Constructor)  {constructor del hash}  Pre{null}  Post{Hash} |

1. Análisis.
   1. Análisis de complejidad temporal de al menos dos de los algoritmos implementados.

public Node<T> pop() {  
  
 Node<T> first = peek();  
  
 if (first == null) {  
 System.*out*.println("Todos los pasajeros han salido del avión");  
 }  
 else {  
 Node<T> nextPop = first.getNext();  
 leader = nextPop;  
 if(nextPop != null) {  
 nextPop.setPrevious(null);  
 }  
 }  
 return first;  
}

R//El método "pop" tiene una complejidad temporal de O(1) porque el tiempo de ejecución del algoritmo no depende del tamaño de la estructura de datos (en este caso, la cola doblemente enlazada).

En otras palabras, independientemente de cuántos elementos tenga la cola, el tiempo de ejecución del método "pop" siempre será constante y no aumentará a medida que la cola crezca. Esto se debe a que las operaciones que realiza el método "pop" (como acceder al primer elemento de la cola, actualizar el puntero "leader" y establecer el nodo anterior del nuevo líder como nulo) son operaciones que se pueden realizar en un tiempo constante. Por lo tanto, la complejidad temporal del método "pop" es O(1).

public void add(T value) throws HeapIsFullException {  
 if (heapSize == heapArray.length) {  
 throw new HeapIsFullException();  
 }  
  
 heapArray[heapSize] = value;  
 heapSize++;  
 siftUp(heapSize - 1);  
}

R//La complejidad temporal total del método "add" será la suma de las complejidades temporales de estas tres operaciones, lo que resulta en O(log n), ya que la comparación y la asignación tienen una complejidad temporal constante y la llamada al método "siftUp" tiene una complejidad temporal logarítmica.

* 1. Análisis de complejidad espacial de al menos dos de los algoritmos implementados.

public T poll() {  
 if (heapSize == 0) {  
 return null;  
 }  
  
 T rootValue = heapArray[0];  
 heapSize--;  
 heapArray[0] = heapArray[heapSize];  
 siftDown(0);  
 return rootValue;  
}

R// La complejidad espacial de este método se divide en dos partes: la complejidad espacial de la operación siftDown y la complejidad espacial de las operaciones adicionales del método. lo que significa que no dependen del tamaño del heap. Por lo tanto, la complejidad espacial total del método poll() es O(log n).

public void enqueue(T item) {  
 if (isEmpty()) {  
 Node<T> node = new Node<>(item);  
 firstInQueue = node;  
 lastInQueue = node;  
 } else {  
 Node<T> nodeToAdd = new Node<>(item);  
 lastInQueue.setNext(nodeToAdd);  
 nodeToAdd.setPrevious(lastInQueue);  
 lastInQueue = nodeToAdd;  
 }  
 size++;  
}

R// La complejidad espacial de enqueue es O(1) solamente en el caso de que la cola esté vacía y se cree un nuevo nodo. En los demás casos, la complejidad espacial es O(n), donde n es la cantidad de elementos ya presentes en la cola, debido a la creación de un nuevo objeto Node para cada elemento adicional.

**TEST TABLES**

**Stack:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la Prueba:** Verificar que los métodos, isEmpty, push, y pop, funcionen correctamente como lo espera el algoritmo | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado** |
| Stack | isEmpty | SetupCase1 | - | El método retorna TRUE. |
| Stack | isEmpty | setupCase1 | Element = “Element” | El método retorna FALSE. |
| Stack | pop | setupCase2 | Element = “Element” | Al sacar el elemento, el Stack quedará vacío. |
| Stack | peek | setupCase1 | Element = “element” | La comparacion de .peek con un String “element” retorna TRUE. |

**Queue:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la Prueba:** Verificar que los métodos, isEmpty, enqueue y dequeue, funcionen correctamente. | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado** |
| Queue | isEmpty | SetupCase1 | - | El método retorna TRUE. |
| Queue | isEmpty | setupCase1 | Element = “Element” | El método retorna FALSE. |
| Queue | enqueue | setupCase2 | Element = “Element1”  Element = “Element2” | Al sacar los 2 elementos de la pila, obtendremos “Element2” y “Element1” en ese orden |
| Queue | dequeue | setupCase2 | Element = “element” | Al usar Dequeue, la pila quedará vacía. |

**HashTable:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la Prueba:** Verificar que los métodos, isEmpty, enqueue y dequeue, funcionen correctamente. | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado** |
| HashTable | insert | SetupCase1 | -element = “element1” | Al buscar el element con su llave, retorna el String ingresado. |
| HashTable | insert | setupCase2 | String = “Element”  Double = 4.56  Boolean = false  Cat = cat | Si se crea la HashTable sin especificar los típos de parametros, no habrá excepciones al ingresar cualquier tipo de elemento. |
| HashTable | search | setupCase2 | Element = “Element1” | Al buscar un element con una llave diferente, retornará NULL, |
| HashTable | search | setupCase3 | Element = “element1”  Element = “element2”  Element = “element3” | Al buscar elementos con sus llaves respectivas, retornará los element encontrados. |