INFORME DE INGENIERIA

LABORATORIO 2

Hippodrome

Presentado por:

Marisol Giraldo Cobo

Presentado al Profesor:

Andrés Aristizábal

Universidad ICESI

Santiago de Cali, marzo 24 del 2020

**INFORME DE INGENIERÍA**

**CONTEXTO DEL PROBLEMA**

Hipódromo puede describirse como una arena donde se disputan carreras de caballos y en donde las personas pueden apostar por quién creen que ganará la carrera. De ahí que, un novedoso hipódromo llamado **El indomable Spirit** en la ciudad de Cali, requiere un sistema que permita el registro de entre siete y diez jinetes con sus respectivos caballos que serán quienes competirán en cada carrera y cuyo orden de salida (de carril) está dispuesto por el orden de inscripción (es decir, el carril número uno corresponde al primer jinete, el número dos al segundo y así sucesivamente).

**PASO 1: IDENTIFICACIÓN DEL PROBLEMA**

**IDENTIFICACIÓN DE NECESIDADES Y SÍNTOMAS:**

* Registrar los caballos con sus corredores.
* Registrar las apuestas dentro de un periodo de tiempo de 3 minutos antes de la carrera.
* Realizar una acción de revancha con los mismos competidores, pero intercambiando su orden de llegada.
* Poder hacer un nuevo evento con otros corredores y caballos nuevos.
* Consultar una apuesta al final de la carrera

**DEFINICIÓN DEL PROBLEMA**

El Hipódromo **El indomable Spirit** en la ciudad de Cali, requiere una aplicación que permita el registro de entre siete y diez jinetes con sus respectivos caballos que serán quienes competirán en cada carrera y cuyo orden de salida (de carril) está dispuesto por el orden de inscripción (es decir, el carril número uno corresponde al primer jinete, el número dos al segundo y así sucesivamente).

**PASO 2: RECOPILACIÓN DE INFORMACIÓN**

*Estructura de Datos:*

Las estructuras de datos en programación son diferentes formas de organizar información para manipular, buscar e insertar estos datos de manera eficiente.

*Complejidad Temporal:*

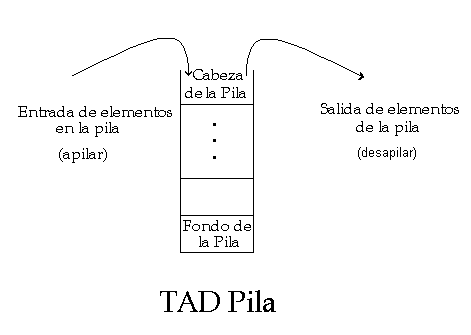
Se denomina complejidad temporal a la función T(n) que mide el número de instrucciones realizadas por el algoritmo para procesar los n elementos de entrada. Cada instrucción tiene asociado un costo temporal.

*Arreglos*

Los arrays son una estructura que almacena los datos un elemento al lado del otro. En la mayoría de los lenguajes de programación esta estructura de datos es de tamaño fijo y no puede guardar datos de diferentes tipos.

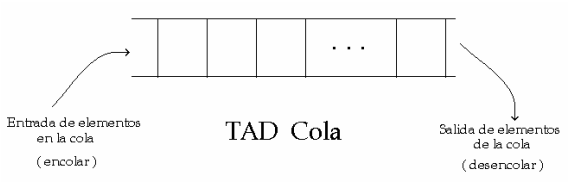
*Listas Enlazadas*

Las listas enlazadas son un tipo de estructura de datos similar a los arrays con la diferencia de que por defecto no tenemos por qué saber la cantidad de elementos que va a contener. Estas listas se componen de nodos los cuales tienen dos atributos: el primero es el ítem o elemento que va a contener este nodo y el segundo atributo es una referencia al siguiente elemento de la lista.



*Estructura de Datos - Lineales FIFO (Queue)*

Esta estructura es otro tipo de lista que nos permite emular el comportamiento de una fila o cola de la vida real donde el primer elemento en ingresar a la fila es el primero en salir, lo que quiere decir que las inserciones (Encolar) se realizan al final y las extracciones (Desencolar) se realizan al frente de la cola, lo cual se conoce como FIFO (First In First Out).



*Estructura de Datos - Lineales LIFO (Stacks*)

Las pilas son un tipo de listas que tienen la particularidad de sólo poder eliminar o insertar en la cima de la lista. A estas acciones se le conocen como apilar y des apilar y conlleva a que el último elemento que ingresa a la pila sea el primero en salir a lo cual se le conoce como LIFO (Last In First Out).

*Eficiencia Algorítmica*

Es usado para describir la cantidad de recursos utilizados por un algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza.

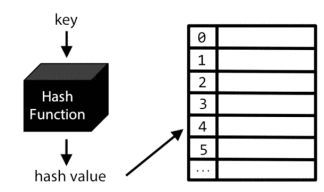
.

*Invariante:*

Es una condición que se sigue cumpliendo después de la ejecución de determinadas instrucciones. Se cumple tanto antes como después de estas instrucciones, permaneciendo sin variación.

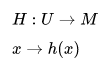
*Tabla de Hash:*

Es una estructura de datos que asocia *llaves* o *claves* con *valores*. La operación principal que soporta de manera eficiente es la *búsqueda*: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una **función hash** en un *hash*, un número que identifica la posición (*casilla* o *cubeta*) donde la tabla hash localiza el valor deseado.



*Función Hash*

Una función hash H es una función computable mediante un algoritmo tal que:



Tiene como entrada un conjunto de elementos, que suelen ser cadenas, y los convierte en un rango de salida finito, normalmente cadenas de longitud fija. Es decir, la función actúa como una proyección del conjunto U sobre el conjunto M. Hay que tener en cuenta que M puede ser un conjunto definido de enteros.

*TAD*

U n Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una especificación independiente de cualquier representación.

TAD = valores + operaciones

La manipulación de un TAD sólo depende de su especificación, nunca de su implementación

*Estructura de Datos – Diccionario*

En un diccionario de datos se encuentra la lista de todos los elementos que forman parte del flujo de datos de todo el sistema. Los elementos más importantes son flujos de datos, almacenes de datos y procesos. El diccionario de datos guarda los detalles y descripción de todos estos elementos, incluyendo nombre, descripción, alias, contenido y organización.

**PASO 3. BÚSQUEDA DE SOLUCIONES CREATIVAS**

**ALTERNATIVA 1: Sensor**

La *Photo Finish*, en una carrera deportiva, es la fotografía de la llegada de los competidores a la línea de meta. Se realiza con una alta precisión que, en caso de ser necesario, ayuda a decidir el ganador.

**ALTERNATIVA 2: Juez de Meta**

El reglamento especifica la necesidad de la presencia de personas que hagan el papel de Peso, Juez de Paddock, Juez de Salida, Juez de Meta y demás personal que se estime necesario para el normal desarrollo de las reuniones de carreras; De ahí, que un juez de meta podría establecer el ganador de la carrera.

**Alternativa 3: Aplicación**

Desarrollar una aplicación en java que permita el registro de entre siete y diez jinetes con sus respectivos caballos que serán quienes competirán en cada carrera y cuyo orden de salida (de carril) está dispuesto por el orden de inscripción (es decir, el carril número uno corresponde al primer jinete, el número dos al segundo y así sucesivamente).

**PASO 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES**

**ALTERNATIVA 1: Sensor**

Si bien es una técnica frecuentemente utilizada en las carreras de caballos, no se cumpliría con los requerimientos de software que el cliente solicita.

**ALTERNATIVA 2: Juez de Meta**

Si bien es una técnica frecuentemente utilizada en las carreras de caballos, no se cumpliría con los requerimientos de software que el cliente solicita.

**Alternativa 3: Aplicación**

Para garantizar el funcionamiento de la aplicación y buenas prácticas de implementación se utilizará las siguientes estructuras de datos para poder llevar a cabo las carreras de caballos:

**Stack (Pila)**

El acceso a los elementos de la pila se realizará siempre sobre un único extremo. Se insertará un nuevo elemento sobre la cima (**push**) y se extraerá el elemento situado en la cima (**pop**). Una forma de ver esta estructura de datos es como una pila de objetos en la que sólo se puede coger el objeto que está en la cima o apilar más objetos sobre la misma, pero los objetos que sostienen la pila no son accesibles pues de otro modo todo se desmoronaría.

Para este proceso se utilizará una función hash que se encargue de agregar los objetos a la pila usando como clave el nombre, la función hash colocará el primer objeto insertado en una fila respectiva de la tabla, y cada posición de la tabla hash tendrá una estructura de datos tipo pila que contenga como máximo de 64 objetos. En caso de que la pila está llena, ocurrirá una colisión y una segunda función hash la colocará en la posición al lado.

**Queue (Colas)**

Los elementos de la cola se añaden y se eliminan de tal manera que el primero en entrar es el primero en salir. La adición de elementos se realiza a través de la operación encolar (**enqueue**), mientras que la eliminación a través de la operación desencolar (**dequeue**). La operación de encolar inserta elementos por un extremo de la cola, mientras que la de desencolar los elimina por el otro.

Para este proceso se utilizará una función hash que se encargue de agregar los objetos a la cola usando como clave el nombre, la función hash colocará el primer objeto insertado en una fila respectiva de la tabla, y cada posición de la tabla hash tendrá una estructura de datos tipo cola que contenga como máximo de 64 objetos. En caso de que la cola esté llena, ocurrirá una colisión y una segunda función hash la colocará en la posición al lado.

**Hash Table (Open Addressing)**

Mediante el direccionamiento abierto se podrá almacenar los registros directamente en el array. Las colisiones se resolverán mediante un *sondeo* del array, en el que se buscarán diferentes localidades del array hasta que el registro sea encontrado o si llega a una casilla vacía, indicando que no existe esa llave en la tabla. Este sondeo podrá resolverse de las siguientes maneras:

**1. Sondeo lineal**

En el que el intervalo entre cada intento es constante (frecuentemente 1).

**2. Sondeo cuadrático**

En el que el intervalo entre los intentos aumenta linealmente (por lo que los índices son descritos por una función cuadrática).

**3. Doble *hasheo***

En el que el intervalo entre intentos es constante para cada registro, pero es calculado por otra función hash. El doble hasheo tiene pobre rendimiento en el caché, pero elimina el problema de aglomeramiento. Este puede requerir más cálculos que las otras formas de sondeo.

**PASO 5: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN**

A screenshot of a cell phone

Description automatically generated

**PASO 6: PREPARACIÓN DE INFORMES Y ESPECIFICACIONES**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | Laboratorio I  Algoritmos y Estructuras de Datos  **Requerimientos Funcionales** |
| Nombre del Proyecto: | | | Hippodrome |
| Presentado por: | | | Marisol Giraldo Cobo – Código: A00246380 |
| Presentado a: | | | Andrés Aristizábal |
| Fecha: | | | Marzo 24 del 2020 |
| **Nombre** | **R1 – Registrar los caballos con sus corredores** | | |
| **Resumen** | Registrar los caballos con sus respectivos corredores | | |
| **Entradas** | | | |
| - | | | |
| **Resultados** | | | |
| Los caballos han sido registrados con sus respectivos corredores | | | |
| **Nombre** | **R2 – Registrar apuesta para una carrera** | | |
| **Resumen** | Registrar la apuesta en un periodo de tiempo de 3 minutos antes de la carrera | | |
| **Entradas** | | | |
| - | | | |
| **Resultados** | | | |
| La apuesta se ha registrado de manera satisfactoria | | | |
| **Nombre** | **R3 – Realizar una acción de REVANCHA** | | |
| **Resumen** | Realizar una acción de revancha con los mismos competidores, pero intercambiando su orden de llegada | | |
| **Entradas** | | | |
| - | | | |
| **Resultados** | | | |
| La acción de REVANCHA ha sucedido de manera satisfactoria | | | |
| **Nombre** | **R4 – Generar nuevo evento de carrera** | | |
| **Resumen** | Permitir realizar un nuevo evento de carrera con corredores y caballos nuevos. | | |
| **Entradas** | | | |
|  | | | |
| **Resultados** | | | |
| El evento se ha generado de manera satisfactoria | | | |
| **Nombre** | | **R4 – Consultar apuesta** | |
| **Resumen** | | Consultar una apuesta al final de la carrera | |
| **Entradas** | | | |
|  | | | |
| **Resultados** | | | |
| Se ha consultado la apuesta | | | |

|  |  |
| --- | --- |
|  | Laboratorio 2  Algoritmos y Estructuras de Datos  Pruebas Unitarias Automáticas |
| Nombre del Proyecto: | Hippodrome |
| Presentado por: | Marisol Giraldo Cobo – Código: A00246380 |
| Presentado a: | Andrés Aristizábal |
| Fecha: | Marzo 24 del 2020 |

|  |  |
| --- | --- |
| **Número de caso de prueba** | 1 |
| **Objetivo** | Comprobar el método de consulta de la estructura de datos “Cola” cuando su frente es nulo |
| **Entradas** | Una cola sin ningún competidor del hipódromo. |
| **Salidas** | Un booleano |
| **Método de validación** | El frente de la cola es nulo |

|  |  |
| --- | --- |
| Número de caso de prueba | 2 |
| Objetivo | Comprobar el método de consulta de la estructura de datos “Cola” cuando su frente no es nulo |
| Entradas | Una cola con los siguientes competidores  [1: { name:”Marisol”, horse:“Centella”}] |
| Salidas | Un booleano |
| Método de validación | Al consultar el nombre del competidor en el frente de la cola, este es igual a Marisol. |

**TAD**

|  |  |
| --- | --- |
| **ADT Stack** | |
| Definition | The ADT Stack is a linear sequence of an arbitrary number of items, along with access procedures. Note that sequence does not imply order. The access procedures permit insertion, deletion and retrieval of elements only at one end of the sequence (the “top”). It’s because of this that is called a last-in-first-out (or LIFO) list.  A stack is either empty, or it consists of a sequence of items. |
| Object Representation |  |
| Invariant | (s.isEmpty() == true ∧ top == NIL) ⊕ (s = {an→an-1→**...**→a1} ∧ top == an) |
| Procedures | * **Constructor operations**   + *createStack()*   //Creates an empty stack  //**post:** A stack with no elements inside has been created   * **Analyzer operations**   + *top()*   //Returns the element at the top of the stack but does not  //modify it.  //**post:** Returns the top element of the stack. If the stack is  //empty then its top is NIL and this is the retrieval.   * + *isEmpty()*   //Determines whether a stack is empty.  //**post:** Returns TRUE if the top of the stack is NIL and FALSE  //otherwise.   * + *size()*   //Returns the size of the stack.  //**post:** Returns the total number of elements in the stack**.**   * **Modifier operations**   + *push(newItem)* |

|  |  |
| --- | --- |
|  | //Adds newItem to the stack.  //**post:** newItem is at the top of the stack and the previous top  //is under it in the same order as before.   * *pop()*   //Removes and returns the top element of the stack.  //**pre:** The stack is not empty.  //**post:** The top is taken off the stack and the new top is the  //element that was under the previously returned element. If  //the stack is empty, it results in an error. |

(aStack.createStack()).size() = 0

(aStack.push(item)).size() = aStack().size() +1

(aStack.pop()).size() = aStack.size() - 1

(aStack.createStack()).isEmpty() = TRUE

(aStack.push(item)).isEmpty() = FALSE

(aStack.createStack()).pop() = ERROR

(aStack.push(item)).top() = item

(aStack.push(item)).pop() = item

(((aStack.push(item1)).push(item2)).pop()).top() = item1

Axioms of the ADT Stack

|  |  |
| --- | --- |
| **ADT Queue** | |
| Definition | The ADT Queue is a linear sequence of an arbitrary number of items, along with access procedures. Note that sequence does not imply order. The access procedures permit insertion of elements only at the back of the sequence. Deletion and retrieval of elements are only performed at the front of the sequence. It’s because of this that is called a  first-in-first-out (or FIFO) list.  A queue is either empty, or it consists of a sequence of items. |
| Object Representation | front = a1 ∧ back = an |
| Invariant | (q.isEmpty() == true ∧ front == back == NIL) ⊕ (q = {a1→a2→**...**→an} ∧  front == a1 ∧ back == an) |
| Procedures | * **Constructor operations**   + *createQueue()*   //Creates an empty queue.  //**post:** A queue with no elements inside has been created.   * **Analyzer operations** |

|  |  |
| --- | --- |
|  | * *front()*   //Returns the element at the front of the queue but does not modify it.  //**post:** Returns the front element of the queue. If the queue is  //empty then its front is NIL and this is the retrieval.   * *isEmpty()*   //Determines whether a queue is empty.  //**post:** Returns TRUE if the front of the queue is NIL and  //FALSE otherwise.   * *size()*   //Returns the size of the queue.  //**post:** Returns the total number of elements in the queue.   * **Modifier operations**   + *enqueue(newItem)*   //Adds newItem to the queue.  //**post:** newItem is at the back of the queue.   * + *dequeue()*   //Removes and returns the front element of the queue.  //**pre:** The queue is not empty.  //**post:** The front is taken off the queue and the new front is the  //element that was behind the previously returned element. If  //the queue is empty, it results in an error. |

(aQueue.createQueue()).size() = 0

(aQueue.enqueue(item)).size() = aQueue().size() +1

(aQueue.dequeue()).size() = aQueue.size() - 1

(aQueue.createQueue()).isEmpty() = TRUE

(aQueue.enqueue(item)).isEmpty() = FALSE

(aQueue.createQueue()).dequeue() = ERROR

Axioms of the ADT Queue

|  |  |
| --- | --- |
| **ADT Hash Table** | |
| Definition | The ADT hash table is a sort of dictionary, which means it stores elements in pairs, a search key and a value. Insertion, deletion and retrieval of items is made by search key and not by value. As in dictionaries, there are no two elements with the same search key. A hash table can be empty.  The ADT Hash Table is an array of elements together with a hash function and access procedures. The access procedures permit insertion, deletion, and retrieval of an item by means of the hash function.  The hash function determines the location in the table for any item, using its search key. The hash function takes a search key and maps it into an |

|  |  |
| --- | --- |
|  | integer array index. |
| Object Representation |  |
| Invariant | h: U→T={0,1,2,**...**,m-1} ∧ m = |T| = |U| |
| Procedures | * **Constructor operations**   + *createHashTable(size)*   //Creates an empty hash table of the specified size.  //**post:** An empty hash table has been created.   * **Analyzer operations**   + *tableRetrieve(searchKey)*   //Retrieves the element corresponding to this searchKey.  //**post:** Retrieves the element that matches this searchKey or  //NIL otherwise.   * + *isEmpty()*   //Determines whether the hash table is empty.  //**post:** Returns TRUE if it is empty or FALSE otherwise.   * + *tableLength()*   //Determines the number of items in the table  //**post:** It returns the total number of items in the hash table.   * **Modifier operations**   + *tableDelete(searchKey)*   //Deletes the item that has a matching searchKey.  //**post:** Deletes the element which search key matches  //searchKey and returns it, otherwise returns NIL.   * + *tableInsert(newItem)*   //Adds newItem to the hash table  //**post:** newItem is stored in the position specified by the hash  //function. |

(aHT.createHashTable(m)).tableLength() = 0

(aHT.tableInsert(item)).tableLength() = aHT.tableLength() +1

(aHT.tableInsert(item)).tableRetrieve(item.searchKey) = item

Axioms of the ADT Hash Table

4. (aHT.tableInsert(item)).tableDelete(item.searchKey) = TRUE

**Paso 7: Implementación**

**BIBLIOGRAFÍA**

¿Cuánto corren los atletas en pista?

<https://www.trichile.cl/?q=Articulo_Cuanto_corren_los_atletas_en_pista>

Estructura de Datos Tablas Hash

<https://cprog.cubava.cu/2018/03/25/tablas-hash/>

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to

Algorithms. Chapters 10 and 11.