**Aplicación del método de la ingeniería para el laboratorio #2: BookStore.**

1. **Identificación del problema**

Se desea realizar una simulación para una librería. El proceso se requerido es el siguiente: Una nueva librería en la ciudad de Cali desea prestar el servicio de venta de libros de una manera rápida y eficiente. Para esto la librería se divide en diferentes secciones en donde los usuarios pueden realizar diferentes actividades como la de consulta de disponibilidad de libros, información completa y detallada de los libros, búsqueda del libro de manera virtual, y elección de los códigos de los libros que desea para posteriormente dirigirse a los estantes donde encontrará los textos escogidos anteriormente por medio de sus respectivos códigos. Por último el cliente podrá dirigirse a la sección de pagos donde en orden del tiempo que tardaron en hacer el proceso de la obtención de los ejemplares para así proceder a cancelarlos.

1. **Recopilación de información**

Librería: establecimiento comercial o tienda que se dedica a la venta de libros. Existen librerías de todo tipo, desde pequeños locales con pocos ejemplares hasta edificios enteros que ofrecen miles de publicaciones. Las librerías pueden pertenecer a una cadena con muchas sucursales o tener un único punto de venta.

Catalogo: un registro que presenta, de manera ordenada, descripciones y datos generales de individuos, objetos, documentos u otras cosas que mantienen algún tipo de vínculo entre sí.

ISBN: Sigla de la expresión inglesa international standard book number, 'número estándar internacional de libro', número de identificación internacional asignado a los libros.

Stands: es el espacio dentro de una feria o salón en el que una empresa expone y presenta sus productos o servicios. Si son de tamaño considerable se le suele llamar también pabellón, así como expositor si es de pequeño tamaño.

Lista ordenada: Son aquellas en las que cada elemento se encuentra precedido por un número, o letra que representa la posición que dicho elemento ocupa en la lista.

Pila: Lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (ultimo en entrar-primero en salir). Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos.

Cola: estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro. También se le llama estructura FIFO, debido a que el primer elemento en entrar será también el primero en salir.

Cola de prioridad:  tipo de dato abstracto similar a una cola en la que los elementos tienen adicionalmente, una prioridad asignada. En una cola de prioridades un elemento con mayor prioridad será desencolado antes que un elemento de menor prioridad. Si dos elementos tienen la misma prioridad, se desencolarán siguiendo el orden de cola.

1. **Búsqueda de soluciones creativas**

Existen diferentes estructuras de datos que nos permiten almacenar una serie de datos, unas más eficientes que otras. A continuación se presentan una serie de alternativas para manejar los datos de la librería.

Alternativa 1: Listas enlazadas.

La lista enlazada es un TDA que nos permite almacenar datos de una forma organizada, al igual que los vectores pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber "a priori" los elementos que puede contener. En una lista enlazada, cada elemento apunta al siguiente excepto el último que no tiene sucesor y el valor del enlace es null. Por ello los elementos son registros que contienen el dato a almacenar y un enlace al siguiente elemento. Los elementos de una lista, suelen recibir también el nombre de nodos de la lista.

Alternativa 2: ArrayList.

La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño. Además permite modificar, insertar y eliminar elementos en cualquier lugar del arreglo.

Alternativa 3: Pila.

Una pila es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de modo que el primer elemento en entrar es el primer elemento en salir (LIFO). Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos. En ésta estructura de datos se puede apilar o des apilar elementos, es decir modificar únicamente el extremo superior de la estructura.

Alternativa 4: Cola.

Una cola (también llamada fila) es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

Las colas se utilizan en sistemas informáticos, transportes y operaciones de investigación (entre otros), donde los objetos, personas o eventos son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento. Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas.

Alternativa 5: Hash table.

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 almacenados a partir de una clave generada. Funciona transformando la clave con una función hash en un hash, un número que identifica la posición donde la tabla hash localiza el valor deseado.

Las tablas hash se suelen implementar sobre vectores de una dimensión, aunque se pueden hacer implementaciones multi-dimensionales basadas en varias claves. Como en el caso de los arrays, las tablas hash proveen tiempo constante de búsqueda promedio O(1), sin importar el número de elementos en la tabla. Sin embargo, en casos particularmente malos el tiempo de búsqueda puede llegar a O(n), es decir, en función del número de elementos.

Las anteriores estructuras de datos serán utilizadas en este caso para:

* Almacenar el catálogo de los libros.
* Simular el carrito de compras.
* Modelar la fila de clientes para cancelar sus productos.
* La forma como se empacan los libros.

1. **Transición de las ideas a los diseños preliminares**

Alternativa 1:

* Tamaño dinámico que implica una mejor optimización de la memoria.
* Los elementos se pueden insertar en una lista indefinidamente.
* las listas enlazadas sólo permiten acceso secuencial a los elementos.
* Las listas enlazadas simples, solo pueden ser recorridas en una dirección.
* Las listas simples requieren la dirección del nodo anterior para insertar o suprimir correctamente.

Alternativa 2:

* Un ArrayList tiene un tamaño dinámico.
* el ArrayList es unidimensional.
* un ArrayList puede cambiar de tamaño según se necesite en tiempo de ejecución.
* admite elementos duplicados y nulos, así como acceso por índice y otros métodos para trabajar con sus elementos.

Alternativa 3:

* En cada momento sólo se tiene acceso a la parte superior de la pila.
* Introduce un elemento en el extremo de la pila.
* Elimina un elemento de la pila (superior extremo).
* Indica si la pila está o no vacía.
* Permite realizar apilamiento, des apilamiento y verificación de vacío.

Alternativa 4:

* Los elementos salen en el mismo orden en que llegan.
* Almacena al final de la cola un elemento.
* Saca de la cola al elemento que se encuentra al frente.
* Se crea la cola vacía.

Alternativa 5:

* El acceso a los datos suele ser muy rápido.
* Debe tener una razón de ocupación no muy elevada.
* Función bien diseñada para evitar que hayan muchas colisiones.
* Necesidad de ampliar el espacio cuando los datos crecen mucho.
* Se suelen emplear listas para recorrer la totalidad de los elementos.

1. **Evaluación y selección de la mejor solución**

Criterios para la evaluación:

Criterio A: Complejidad (peor caso).

* [8] complejidad constante.
* [7] complejidad logarítmica.
* [6] complejidad radical.
* [5] complejidad lineal.
* [4] complejidad n(log n).
* [3] complejidad polinómica.
* [2] complejidad exponencial.
* [1] complejidad factorial.

Criterio B: Eficiencia en manejo de datos

* [2] Eficiente para muchos datos.
* [1] Eficiente para pocos datos.

Criterio C: Dificultad de implementación.

* [3]Facilidad de implementación.
* [2]Normalidad en la implementación.
* [1]Dificultad para la implementación.

Evaluando los algoritmos bajo los anteriores criterios obtenemos la siguiente tabla de calificaciones:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Criterio A | Criterio B | Criterio C | Total |
| Listas enlazadas | Inserción O(n^2) 2 | Se adquieren posiciones de memoria a medida que se necesitan y se liberan cuando ya no se requieren 2. | Se requiere recorrerlas. 2 | 6 |
| ArrayList | En promedio O(k) 8 | Se debe recorrerlos. 1 | En ocasiones se necesitan más de un solo recorrido. 1 | 10 |
| Pila | Todas las operaciones de la pila se pueden manejar en tiempo constante.  8. | No se deben recorrer todos los datos.  2 | Posee tres métodos de modificación; insertar, modificar y eliminar. 2 | 12 |
| Cola | Las operaciones pueden ser ejecutadas en O(1).  8 | Se pude acceder al elemento más antiguamente insertado.  1 | Posee por defecto tres operaciones naturales.  3 | 12 |
| Hash table | En el peor de los casos y utilizando una buena función: O(1). 8 | Se puede establecer y modificar el tamaño de la tabla. 1 | Facilidad de acceder a los datos.  2 | 11 | |

Analizando cada uno de los casos, llegamos a la conclusión de que las alternativas 3,4 y 5 son las mejores a implementar, debido a su eficiencia algorítmica.

1. **Diseños preliminares.**

1. Requerimientos funcionales:

|  |  |
| --- | --- |
| **R1** | **Mostrar Catalogo de libros.** |
| **Resumen** | Permite al usuario revisar la disponibilidad de los libros y hacer consultas sobre ellos. |
| **Entrada** | Ninguna. |
| **Salida** | Stands de libros con sus respectivos códigos ISBN. |

|  |  |
| --- | --- |
| **R2** | **Seleccionar los libros**. |
| **Resumen** | El usuario selecciona los libros de los catálogos y obtiene sus respectivos códigos (ISBN), almacenándolos en una lista. |
| **Entrada** | Código de los libros. |
| **Salida** | Lista con los libros seleccionados. |

|  |  |
| --- | --- |
| **R3** | **Buscar libro.** |
| **Resumen** | Mediante el ISBN, el usuario podrá consultar el lugar en donde se encuentra dicho libro. Si el libro está agotado, no aparecerá en la lista final del usuario. |
| **Entrada** | Código de los libros. |
| **Salida** | Stand donde se encuentra el libro. |

|  |  |
| --- | --- |
| **R4** | **Añadir los libros al canasto.** |
| **Resumen** | Permite al usuario colocar los libros uno encima del otro en el orden en que los recoge de los stands. Orden suministrado previamente. |
| **Entrada** | Lista de libros seleccionados. |
| **Salida** | Pila con los libros tomados de los stands. |

|  |  |
| --- | --- |
| **R5** | **Pago de libros.** |
| **Resumen** | Dispone a lo usuarios en una única cola la cual se ordena de acuerdo al tiempo empleado por cada cliente en su proceso de compra. |
| **Entrada** | Usuario que realizará el pago. |
| **Salida** | Cola de prioridad de usuarios ordenada por tiempo. |

|  |  |
| --- | --- |
| **R6** | **Recepción del pago.** |
| **Resumen** | Hay varias cajas que se encargan de atender la única cola de usuarios en orden de llegada. |
| **Entrada** | Cola de usuarios. |
| **Salida** | Valor de pago total para cada usuario atendido. |

|  |  |
| --- | --- |
| **R7** | **Empacar libros.** |
| **Resumen** | Los libros serán empacados en el orden que estén en el carrito de compras. Es decir, El último en ser puesto en el canasto será el primero en ser empacado, y el primer libro que se pone en el canasto será el último en ser empacado. |
| **Entrada** | Lista de un usuario |
| **Salida** | Pila con los libros comprados. |

Requerimientos no funcionales.

R1: Ordenar la lista de libros para cada cliente de manera que tenga que recorrer el menor camino posible por los stands.

R2: Contabilizar el tiempo que demora el proceso de compra de cada cliente con respecto a la cantidad de libros que desea comprar.

2. Diseño de los TADs

|  |  |
| --- | --- |
| **TAD** | **Hashtable** |
| Definición gráfica. | |
| ***Invariantes***   * Factor de carga < 40%. * No deben existir colisiones y si se dan se deben manejar con la finalidad que no afecten la eficiencia de la estrucutura. | |
| ***Operaciones:***  CreateHashtable() 🡪 Hashtable🡪 Constructor  removeKey() 🡪 E 🡪 Hashtable  put() 🡪 E, E 🡪 Hashtable  searchKey() 🡪 E🡪 boolean  searchValue() 🡪 E 🡪 boolean  isEmpty() 🡪 Hashtable 🡪 boolean  getKey() 🡪 Hashtable 🡪 E  setKey() 🡪 E🡪 Hashtable  getValue() 🡪 Hashtable 🡪 E  setValue() 🡪 E🡪 Hashtable  getTamanio() 🡪 Hashtable 🡪 integer  setTamanio() 🡪 Integer 🡪 Hashtable  getArray() 🡪 Hashtable 🡪 Arreglo  setArray() 🡪 Arreglo –> Hashtable | |
| * *Pre:**Las variables están declaradas* * *Post:**Las variables han sido instanciadas* * *Def:**Constructor de la clase.*   *public Hashtable()*   * *Pre:**La llave está declarada e instanciada.* * *Post:**la llave encontrada ha sido eliminada.* * *Def:**El método se encarga de eliminar una llave.*   *public void removeKey(E k)*   * *Pre:**La llave y el valor, han sido declarados e instanciados.* * *Post****:*** *El objeto ha sido agregado a la hashtable.* * *Def:**El método se encarga de agregar un objeto a la tabla hash.*   *public void put(E k, E v)*   * *Pre:**la tabla hash tiene objetos.* * *Post:**Se ha verificado si la llave está en la tabla hash.* * *Def:**Se encarga de verificar si una llave se encuentra dentro de la tabla hash.*   *public boolean searchKey(E k)*   * *Pre:**la tabla hash tiene objetos.* * *Post:**Se ha verificado si el valor está en la tabla hash.* * *Def:**Se encarga de verificar si una valor se encuentra dentro de la tabla hash.*   *public boolean searchValue(E v)*   * *Pre:**La tabla hash ha sido creada, vacía.* * *Post:**Se ha verificado si la tabla hash no tienen ningún componente.* * *Def:**Verifica si la tabla hash no tiene elementos.*   *public boolean isEmpty()* | |

|  |  |
| --- | --- |
| **TAD** | **Pila** |
| Definición Gráfica | |
| ***Invariantes***  Conserva el orden de entrada y salida de elementos, LIFO (Last in- First out). | |
| ***Operaciones:***  *CreateStack() 🡪 Stack🡪 Constructor*  *getTop() 🡪 E🡪 E*  *longitude() 🡪E 🡪 Integer*  *isEmpty() 🡪 E 🡪 boolean*  *remove() 🡪 E 🡪 Player* | |
| * *Pre:**La pila ha sido inicializada.* * *Post:**Se ha agregado un elemento a la pila* * *Def:**El método se encarga de insertar un elemento en la pila, el cuál estará en el tope (cima).*   *public void CreateSlack(E e)*   * *Pre:**la pila ha sido inicializada.* * *Post:**Se ha retornado el elemento top.* * *Def:**El método retorna el elemento que se encuentra en la cima de la pila. El elemento top.*   *public E getTop()*   * *Pre:**la pila está inicializada.* * *Post:**Se ha retornado el número de elementos de la pila.* * *Def:**El método retorna el tamaño de la pila, el cual corresponde al número de elementos contenidos.*   *public int longitude()*   * *Pre:**La pila está inicializada.* * *Post:**Se ha verificado que la pila tiene por lo menos un elemento.* * *Def:**El método retorna un booleano que determina si la pila contiene elementos.*   *public boolean isEmpty()*  *Pre: La pila está inicializada y no vacía.*  *Post: La pila contiene un elemento menos*  *Def: Retorna el elemento del tope de la pila y lo borra. El que estaba debajo del tope pasa a ser el top ahora.*  *Public E remove()* | |

|  |  |
| --- | --- |
| **TAD** | **Cola** |
| Definición gráfica | |
| ***Invariantes***  Conserva el orden de entrada y salida de elementos, FIFO (First in- First out). | |
| ***Operaciones:***  *Queue 🡪 Constructor*  *Longitude()* ***🡪*** *E🡪 integer*  *isEmpty() 🡪 E🡪 boolean*  *Queue() 🡪 E🡪 boolean*  *Dequeue() 🡪E 🡪 boolean*  *remove() 🡪 E 🡪 boolean* | |
| * *Pre:**La cola ha sido inicializada.* * *Post:**Se ha agregado un elemento a la cola* * *Def:**El método se encarga de insertar un elemento en la cola, el cuál estará en la cabeza.*   *public void Queue(E e)*   * *Pre:**la cola ha sido inicializada.* * *Post:**Se ha retornado el elemento cabeza de la cola.* * *Def:**El método retorna el elemento que se encuentra en la cima de la pila. El elemento top.*   *public E deQueue()*   * *Pre:**la cola está inicializada.* * *Post:**Se ha retornado el número de elementos de la cola.* * *Def:**El método retorna el tamaño de la cola, el cual corresponde al número de elementos contenidos.*   *public int longitude()*   * *Pre:**La cola está inicializada.* * *Post:**Se ha verificado que la cola tiene por lo menos un elemento.* * *Def:**El método retorna un booleano que determina si la cola contiene elementos.*   *public boolean isEmpty()*  *Pre: La cola está inicializada y no vacía.*  *Post: La cola contiene un elemento menos*  *Def: Retorna el elemento de la cabeza de la cola y lo borra.*  *Public E remove()* | |

3. Diseño de los casos de pruebas unitarias

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DISEÑO DE PRUEBAS UNITARIAS** | | | | |
| CLASE | METODO | ESCENARIO | ENTRADA | SALIDA |
| HashTable | Put() | Creacion de un objeto de tipo hash, de un arreglo con llaves y con valores | Agregar un objeto a la hash con una respective k y valor | El objeto se agregó satisfactoriamente |
| HashTable | Remove() | Creacion de un objeto de tipo hash, de un arreglo con llaves y con valores | Eliminar un objeto dado un valor unoco key | El objeto fue eliminado o ya no existe |
| HashTable | Remove() pull() Empty() | Creacion de una hashtable para comprobar los invariantes | Un arreglo con muchos elementos y retornando | Todos los invariantes se cumplen en la hash |
| Stack | Push() | Creación de un objeto de tipo Stack | Un arreglo con valores que repersentan un entero | Agrega un element dentro de la Stack |
| Stack | Empty() | Creacion de un objeto de tipo Stack | Dos arreglos de con diversos valores y un valor de dichos arreglos | La pila esta vacia representada con un valor booelano |
| Stack | Push() pop() | Creacion de un objeto de tipo Stack para probar que el stack es una estructura lifi | Arreglos de un tipo char que representen oraciones | El stack si se comporta como una estructura lifi |
| Queue | Offer() | Creación de un objeto de tipo Queue generico | Recibe un arreglo que contiene diversos valores a agregar | Agrega satisfactoriamente elementos a la cola |
| Queue | Empty() | Creación de un objeto de tipo Queue generico | Un conjunto de numeros generados aletoriamente para verificar si esta vacia o no | La cola esta vacia o el elemento |
| Queue | Pull() | Creación de un objeto de tipo Queue generico |  | Retorna y borra la cabeza de la queue |
| Queue | Peek() | Creación de un objeto de tipo Queue generico |  | Retorna la cabeza de la queue o null si esta vacia |

**Implementación**

<https://github.com/Jhusseth/Library.git>

**Webgrafia**

* [https://docs.google.com/document/d/1gawsUhajQ-ubJY1nDFI4y0TmwLO4NdyJp-0u6Y9ZQu4/edit#](https://docs.google.com/document/d/1gawsUhajQ-ubJY1nDFI4y0TmwLO4NdyJp-0u6Y9ZQu4/edit)
* <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT1oKweFaYxigGB_xqMU7rWm8Tt7HXkVAqlwMMq72bRAN0gwDIT>
* <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSMohd4Qb8Ufs2OualVpx7CTPUknjEhcLp_GGengV6nySHAxdon>
* <https://definicion.de/libreria/>
* <https://definicion.de/catalogo/>
* <https://es.wikipedia.org/wiki/Estand>
* <http://euler.mat.uson.mx/~havillam/curso_html/listas.html>
* <https://es.wikipedia.org/wiki/Pila_(informática)>
* <https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)>
* <https://es.wikipedia.org/wiki/Cola_de_prioridades>