ICESI
Tercer semestre de 2022

Profesor: Aristizabal
Participantes: Valderruten Soler Fabiana & Valencia Valencia Andy

Tarea integradora I

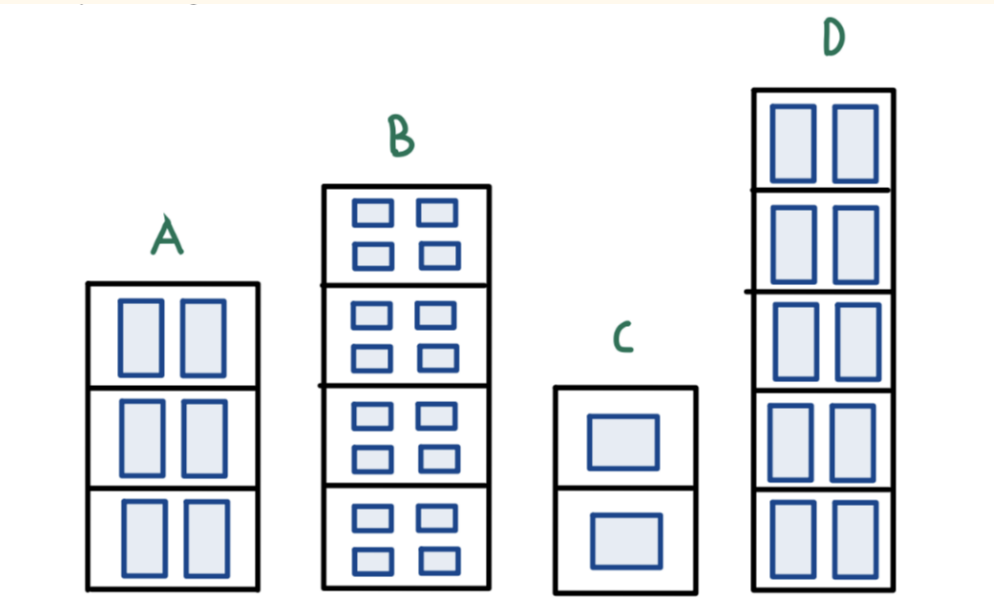
computación & estructuras discretas

Enunciado

- Discreet Guys Inc. de la ciudad de Cali decide contratar a su equipo para simular el funcionamiento de los ascensores de los nuevos edificios que van a construir en el reciente lote que adquirieron cerca de la universidad ICESI.

- Se tiene pensado construir diferentes tipos de edificios inteligentes. Teniendo esto en cuenta, su equipo debe considerar diferentes entradas con diversos casos de prueba para poder satisfacer las necesidades de la empresa.

- Estos edificios contarán con varias oficinas en cada piso, siendo el número de ellas igual por cada piso del edificio al que se esté referenciando en la entrada. Además, cada oficina contará con un número que la identifica en orden ascendente a medida que se desciende por el edificio, es decir, si hay dos pisos y dos oficinas por piso entonces en el piso 2 estarán las oficinas 1 y 2, y en el piso 1 estarán las oficinas 3 y 4. Por último, en cada piso de un edificio se encontrará un número determinado de personas (este puede variar por piso).



Método de ingeniería

Los ingenieros tienden a tratar los problemas de manera única, éste método de ingeniería difiere de muchos otros métodos usados por el resto de profesionales :

1. Identificación del problema.
2. Recopilación de la información necesaria.
3. Búsqueda de soluciones creativas.
4. Paso de las ideas a los diseños preliminares.
5. Evaluación y selección de la solución.
6. Preparación de informes y especificaciones.
7. Implementación del diseño.

Contexto problemático : La universidad ICESI desea construir nuevos edificios llenos de oficinas en un lote que adquieren cerca a la universidad. Discreet Guys Inc decide contratar a su equipo que simule el funcionamiento de los ascensores de los edificios.

Desarrollo de la solución: Para dar una solución al problema el equipo de trabajo usa el método de la ingeniería siguiendo un enfoque sistemático y acorde con la situación de la problemática planteada.

Necesidades:

- ☐ Aún no existe un programa que simule el movimiento entre pisos de los ascensores que conectan a las oficinas de cada edificio.
- ☐ La solución debe mostrar los datos de las ubicaciones iniciales y finales de cada usuario.
- ☐ El programa debe ser eficiente para cada edificio.

1. ***Identificación del problema*** : Icesi requiere de un módulo de software que permita mostrar el comportamiento o traslaciones que hagan los usuarios al cambiar de oficina y piso por medio del ascensor .
2. ***Recopilación de la información necesaria :*** Con el fin de tener más claridad en los conceptos involucrados se hace una búsqueda de información y experiencias del comportamiento de un ascensor habitual. Para realizar una solución más óptima y que cumpla con las competencias del curso, se ha estudiado y trabajado el uso de queues, stacks y tablas hash. Incluso es importante crear casos de prueba que simulan las entradas comunes del programa y así poder comprobar el buen comportamiento del software en solución.

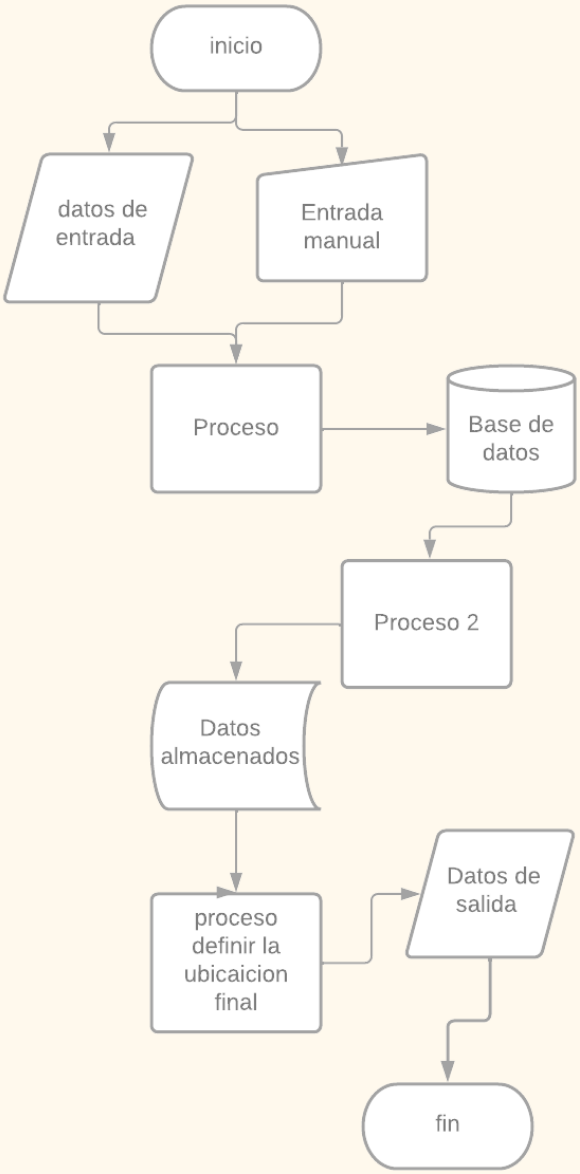
Queues: Una cola es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pull por el otro. También se le llama estructura FIFO, debido a que el primer elemento en entrar será también el primero en salir.
Stacks: es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»). Esta estructura se aplica en multitud de supuestos en el área de la informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos.
Tablas hash: Una tabla hash, matriz asociativa, hashing, mapa hash, tabla de dispersión o tabla fragmentada 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.

3. ***Búsqueda de soluciones creativas :*** El equipo de trabajo ha ideado un programa que pida como entrada en la primera línea la cantidad de edificios que se van a construir. En la segunda línea pide datos como : el identificador del edificio, este puede ser una selección por medio de un menú o sencillamente se lee el identificador como letra, sea A,B,C,D etc..; en esta misma línea de entrada , pero con un espacio de diferencia, se espera pedir la cantidad de usuarios que estará en dicho edificio 0<x<100 . El siguiente número es la cantidad de pisos que contiene el edificio, la línea termina con el número de oficinas que tiene cada piso. A partir de la segunda se espera recibir los datos de cada usuario inicial; estos serían los nombres seguidos del piso en el que están ubicados y de este se acompaña el número de oficina a la que se dirige la persona.
   Una de las ideas que ha tenido el equipo de trabajo, ha sido usar los recursos de java como Queues, tablas hash y Stacks para almacenar los datos.
   Finalmente se planea que el programa arroje un listado de los datos en los que se ha movilizado el usuario de cada edificio. Además se planea hacer y entregar el test para cada estructura, así se comprueba el buen funcionamiento.

4. ***Transformación de las ideas a diseños preliminares :*** Lo primero que se hace es descartar las ideas que no son factibles. En este sentido no se descarta ninguna, ya que el equipo de trabajo es pequeño y cada idea ha sido bien planteada desde el inicio, hablando del comportamiento del programa. Si se tratase del diseño, podrían existir algunas modificaciones, entre la creación de menús para el ingreso de datos o en la presentación de los outputs.

5. ***Evaluación y selección de la solución:*** A medida que evoluciona el proceso de ingeniería y el equipo ha trabajado en la planeación y pruebas de las ideas del software. Se ha llegado a la conclusión que se eligieron las ideas ya planteadas, pero habrá un cambio, los tests que se realizarán para evaluar las estructuras, ya no serán entregables.

6. ***Preparación de informes y especificaciones:***
   Diagrama de flujo basándose en el diseño.



7. ***Implementación del diseño:*** Implementación en un lenguaje de programación.
   lista de tareas para implementar:
   1. validar los datos ingresados
   2. crear los edificios
   3. crear los objetos
   4. crear las estructuras de datos: Stacks, queues, tablas hash

5. identificar el recorrido del usuario (el punto de llegada)
6. mostrar dónde se encuentran los usuarios al finalizar el programa.

código para crear una estructura, colas:

```
package model.data_structures;
import model.interfaces.Intercolas;
import java.util.NoSuchElementException;
public class Colas<T> implements Intercolas<T> {
   private Nodos<T> front;
   private Nodos<T> rear;
   private int size;

      public Colas() {
      front = null;
      rear = null;
      size = 0;
   }
   public void enqueue(T data) {
      Nodos<T> element = new Nodos<>(data);
      if (front == null) {
         rear = element;
         front = element;
      } else {
         rear.setNext(element);
         rear = element;
      }
      size++;
   }
   public T dequeue() {
      Nodos<T> trash = front;
      if (front == null) {
         throw new NoSuchElementException("Can't dequeue
from an empty queue");
      } else if (front == rear) {
         front = rear = null;
      } else {
         front = front.next();
      }
      size--;
      return trash.data();

   }
   public T front() {
      return front.data();
   }

    public boolean isEmpty() {
      return size == 0;
   }

    public int size() {
      return size;
   }


   public String toString() {
      StringBuilder sb = new StringBuilder();
      Nodos<T> head = front;
      sb.append("(");
      String prefix = "";
      while (head != null) {
         sb.append(prefix).append(head.data());
         prefix = ", ";
         head = head.next();
      }
      sb.append(")");
      return sb.toString();
   }
   public Colas<T> reverse() {
      Stack<T> aux = new Stack<>();
      Colas<T> reversed = new Colas<>();
      while (!isEmpty()) {
         aux.push(dequeue());
      }
      while (!aux.isEmpty()) {
         reversed.enqueue(aux.pop());
      }
      return reversed;
   }

   public void toQueue(T[] array) {
      front = null;
      rear = null;

      for (T var : array) {
         enqueue(var);
      }
   }
}
```

Creación de los objetos, Personas o usuarios :

```
package model.objects;

public class Personas {
   private String nombre;
   private int posicion;
   private int oficina;

   public Personas(String nombre, int posicion, int oficina) {
      this.nombre = nombre;
      this.posicion = posicion;
      this.oficina = oficina;
   }

   public String getNombre() {
      return nombre;
   }

   public int getPosicion() {
      return posicion;
   }

   public int getOficina() {
      return oficina;
   }
}
```

Análisis de complejidad temporal

For (int i = 0; i < buildings.length; i++) {	n+1
If (buildings [i].getNombre()== in)	C(n)
Return 1;	1
Return -1	1

(n+1) + c(n) + 1 + 1 = n + cn + 3 = 2n + 3 = O(n) Complejidad Lineal

For (int i = 0; i < countPersona; i++) {	n+1
String[] persona = br.readLine().split(“ ”);	n
Personas p = new Personas(persons[0], Integer.parseInt(personas[1],Integer.parseInt(person[2]));	n
Colas.enqueue(p)	C(n)

3n + cn + 1 = 4n + 1 = O(n) Complejidad Lineal

Análisis de complejidad espacial

	input	Aux	Out
For (int i = 0; i < countPersona; i++) {	1	n	0
String[] persona = br.readLine().split(“ ”);		1	
Personas p = new Personas(persons[0], Integer.parseInt(personas[1],Integer.parseInt(person[2]));		n	

Colas.enqueue(p)			

2n + 2 = O(n)

	input	Aux	Out
For (int i = 0; i < countPerson; i ++) {	1	n	
ColasPrioritarias.offer(Colas.dequeue());			

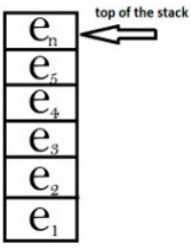
O(n)list

Especificación de requerimientos y diseño

Cliente	Discreet guys inc
Usuario	La universidad icesi, los ingenieros que acompañan el proyecto
Requerimientos funcionales	R1: Recopilar la información de cada edificio, pisos, oficinas y ubicación inicial de cada persona. R2: Operar hacia dónde se dirige cada persona. R3: Dirigir el ascensor a cada piso en base al orden de pulsación del botón.
Contexto del problema	Se construyen nuevos edificios en Icesi y se requiere de un programa que simule los movimientos de un ascensor.

Requerimientos no funcionales	simulación eficiente. Que tenga en cuenta cada movimiento de cada edificio. que las salidas sean específicas.

TADS

Stack ADT
 $Stack = \ll e_1, e_2, e_3, \dots, e_n >, top >$
$\{inv:\ 0 \leq n\ \wedge\ Size(Stack) = n\ \wedge\ top = e_n\}$
$Stack \rightarrow Stack$ $push:\ Stack\ x\ Element \rightarrow Stack$ $pop:\ Stack \rightarrow Stack$

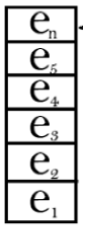
$\boldsymbol{Stack}$
Builds an empty stack $\{\boldsymbol{pre}: \boldsymbol{null}\}$ $\{\boldsymbol{post}: \boldsymbol{Stack\ s} = \emptyset\}$ 

$\boldsymbol{push}$
Adds the new element e to stack s $\{\boldsymbol{pre}: \boldsymbol{Stack\ s} = < \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_n} > \boldsymbol{and\ element\ e\ or\ s} = \emptyset \boldsymbol{\ and\ element\ e}\}$ or $\{\boldsymbol{post}: \boldsymbol{Stack\ s} = < \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_n}, \boldsymbol{e} > \boldsymbol{or\ s} =< \boldsymbol{e} >\}$ or

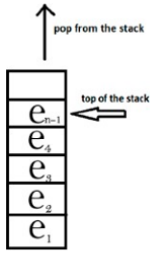
pop

Extracts from the stack s; the most recently inserted element.

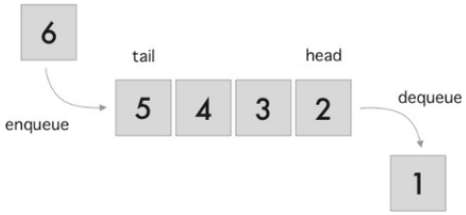
$\{pre: Stack\ s \neq \emptyset\ i.e.\ s = < e_1, e_2, e_3, \dots, e_n >\}$



$\{post: Stack\ s = < e_1, e_2, e_3, \dots, e_{n-1} >\}$



Queue ADT



$Queue = \ll e_1, e_2, e_3, \dots, e_n, front, back >$

$\{inv: 0 \leq n\ \wedge\ Size(Queue) = n\ \wedge\ front = e_1\ \wedge\ back = e_n\}$

$Queue \rightarrow Queue$

$enqueue: Queue\ x\ Element \rightarrow Queue$

$dequeue: Queue \rightarrow Element$

Queue

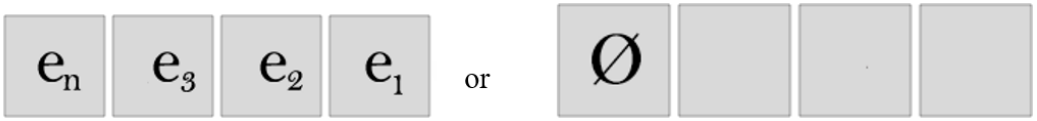
Builds an empty queue

$\{\boldsymbol{pre}: \boldsymbol{null}\}$
$\{\boldsymbol{post}: \boldsymbol{Queue\ q} = \emptyset\}$

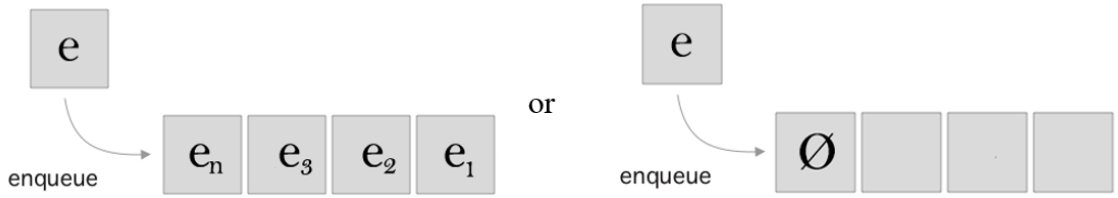
enqueue

Insert a new element e to the back of the queue q

$\{\boldsymbol{pre}: \boldsymbol{Queue\ q} = < \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \ldots, \boldsymbol{e_n} > \boldsymbol{and\ element\ e\ or\ q} = \emptyset \boldsymbol{\ and\ element\ e}\}$



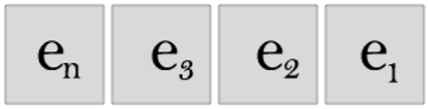
$\{\boldsymbol{post}: \boldsymbol{Queue\ q} =< \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \ldots, \boldsymbol{e_n}, \boldsymbol{e} > \boldsymbol{or\ q} = < \boldsymbol{e} >\}$



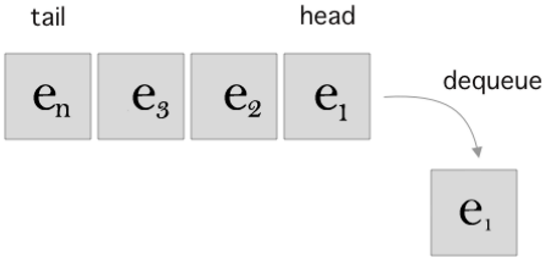
dequeue

Extracts the element in Queue q's front

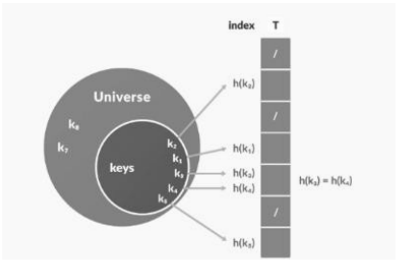
$$\{pre: Queue\ q \neq \emptyset, i.e.\ q = < e_1, e_2, e_3, \dots, e_n >\}$$



$$\{post: Queue\ q = < e_1, e_2, e_3, \dots, e_{n-1} > and\ element\ e_1\}$$



Hash Table ADT



$$HashTable = \ll < k_0, v_0 >, < k_1, v_1 >, < k_2, v_2 >, \dots, < k_n, v_n > \gg$$

$$\{inv: k_0 \in U \wedge h(k) \in A \mid \forall_k\ h(k) \neq A(v_0) \wedge h(k) \neq A(v_1)\}$$

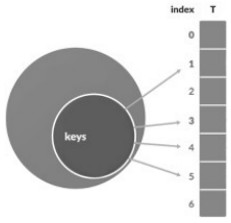
$$HashTable \rightarrow HashTable$$
$$insert: HashTable\ x\ Key\ x\ Value \rightarrow index$$
$$delete: HashTable\ x\ Key \rightarrow HashTable$$
$$search: HashTable\ x\ Key \rightarrow Value$$

HashTable

Creates an empty hash table

$\{\boldsymbol{pre}: \boldsymbol{null}\}$

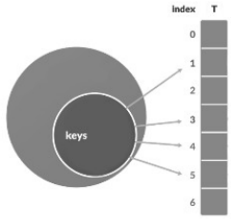
$\{\boldsymbol{post}: \boldsymbol{HashTable\ ht = \theta}\}$



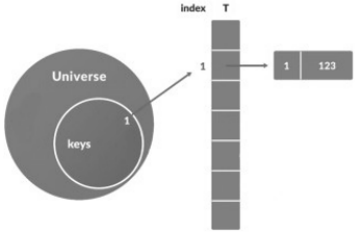
Insert

Inserts a new item into a table in its proper sorted, order according to the new item's search key.

$\{\boldsymbol{pre}: \boldsymbol{hashTable\ h = \ll < k_0, v_0 >, \ldots, < k_n, v_n > \gg\ or\ h = \theta\ and\ Key\ k \neq \theta}\}$



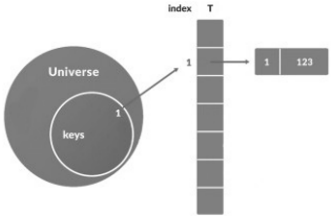
$\{\boldsymbol{post}: \boldsymbol{hashTable\ h = \ll < k_0, v_0 >, \ldots, < k_n, v_n >, < k, v > \gg\ or\ h =< k, v >}\}$



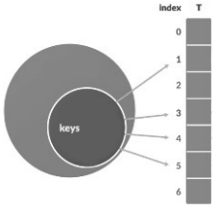
Delete

Deletes an item with a given search key from the table.

$\{\boldsymbol{pre}: \boldsymbol{hashTable\ h} = \ll < \boldsymbol{k_0}, \boldsymbol{v_0} >, \dots, < \boldsymbol{k_n}, \boldsymbol{v_n} > \gg \boldsymbol{and\ Key\ k} \neq \boldsymbol{\theta}\}$



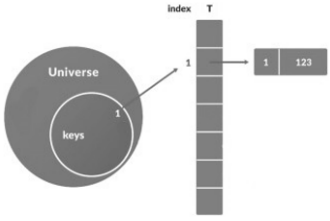
$\{\boldsymbol{post}: \boldsymbol{hashTable\ h} = \ll < \boldsymbol{k_0}, \boldsymbol{v_0} >, \dots, < \boldsymbol{k_n}, \boldsymbol{v_n} >, \gg \boldsymbol{or\ h} = \boldsymbol{\theta}\ \}$



Search

Retrieves an item with a given search key from a table.

$\{\boldsymbol{pre}: \boldsymbol{hashTable\ h} \neq \boldsymbol{\theta}\ \boldsymbol{and\ Key\ k} \neq \boldsymbol{\theta}\}$



$\{\boldsymbol{post}: \boldsymbol{hashTable\ h} \neq \boldsymbol{\theta}\ \boldsymbol{and\ return} < \boldsymbol{k}, \boldsymbol{v} >\ \}$