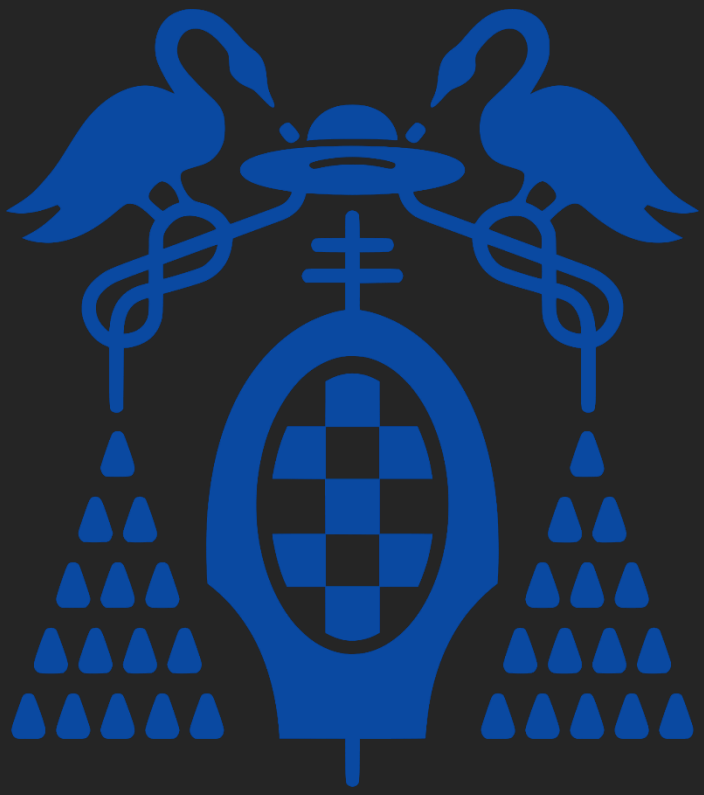
ESTRUCTURA DE DATOS

Gestión Pedidos Amazon



Tian Duque Rey - 03210649X

2023-2024

Índice

1. Especificación concreta de la interfaz

En esta primera parte se incluye en la especificación las principales estructuras de datos estudiadas en clase: Pilas, Colas, Listas. Estas tres estructuras básicas nos ayudaran a modelar el funcionamiento de un sistema de gestión y envió de pedidos de Amazon.

a) PILA

La estructura Pila almacena los paquetes reproduciendo el proceso de empaquetado, donde los paquetes se almacenarán en una única pila LIFO. La especificación se define: NodoPila.hpp, NodoPila.cpp, Pila.hpp, Pila.cpp.

espec PILA[ELEMENTO]
usa BOOLEANOS
parametro formal
generos elemento
fparametro
generos pila
{crear una pila vacía}
pvacía: → pila
{poner un elemento en la pila}
apilar: elemento pila → pila
{quitar un elemento de la pila}
parcial desapilar: pila → pila
{observar la cima de la pila}
parcial cima: pila → elemento
{para ver si la pila está vacía}
vacía?: pila → bool

var p: pila; x: elemento
{Primera forma: utilizando las generadoras del tipo}
ecuaciones de definitud
Def(desapilar(apilar(x,p)))
Def(cima(apilar(x,p)))
{Segunda forma: utilizando propiedades de los datos}
ecuaciones de definitud
vacía?(p) = F ⇒ Def(desapilar(p))
vacía?(p) = F ⇒ Def(cima(p))
ecuaciones
desapilar (apilar(x,p)) = p
cima (apilar(x,p)) = x
vacía? (pvacía) = T
vacía? (apilar(x,p)) = F
fespec

b) COLA

La estructura Cola representa las zonas de estación ordenada de empaquetado, cumpliendo la funcionalidad de procesamiento de los paquetes**.** Está compuesta por: NodoCola.hpp, NodoCola.cpp, Cola.hpp, Cola.cpp

espec COLA[ELEMENTO]
usa BOOLEANOS
parametro formal
generos elemento
fparametro
generos cola
{crear una cola vacía}
cvacía: → cola
{poner un elemento en la cola}
añadir: elemento cola → cola
{quitar un elemento de la cola }
parcial eliminar: cola → cola
{ver el principio de la cola}
parcial primero: cola → elemento
{ver si la cola está vacía}
vacía?: cola → bool var c: cola; x: elemento
{Primera opción: indicando qué forma deben tener los datos}
ecuaciones de definitud
Def(eliminar(añadir(x,c)))
Def(primero(añadir(x,c)))
{Segunda opción: con las propiedades que tienen que cumplir los datos }
ecuaciones de definitud
vacía?(c) = F ⇒ Def (eliminar(c))
vacía?(c) = F ⇒ Def (primero(c))
ecuaciones
eliminar(añadir(x,cvacía)) = cvacía
vacía?(c)=F ⇒ eliminar(añadir(x,c)) = añadir(x,eliminar(c))
primero(añadir(x,cvacía)) = x
vacía?(c)=F ⇒ primero(añadir(x,c)) = primero(c)
vacía?(cvacía) = T
vacía?(añadir(x,c)) = F
fespec

c) LISTA

Por último, la estructura dinámica Listas permiten simular el envío de paquetes que serán almacenados según su prioridad en un área de postempaquetado. Está compuesta por NodoLista.hpp, NodoLista.cpp, Lista.hpp, Lista.cpp

espec LISTA+[ELEMENTO]

usa LISTA[ELEMENTO], NATURALES2

operaciones

{Ver el dato dada una posición, insertar, modificar o borrar}

parcial _ [_]: lista natural → elemento

parcial insertar: elemento lista natural → lista

parcial modificar: elemento lista natural → lista

parcial borrar: lista natural → lista

{Ver si un dato está en la lista, y en qué posición se encuentra}

está?: elemento lista → bool

buscar: elemento lista → natural var n : natural x,y : elemento l : lista

ecuaciones de definitud

$(1 \leq i \leq long(l)) = T \Rightarrow Def(l[i])$

$(1 \leq i \leq long(l)) = T \Rightarrow Def(modificar(x,l,i))$

$(1 \leq i \leq long(l)) = T \Rightarrow Def(borrar(l,i))$

$(1 \leq i \leq long(l)+1) = T \Rightarrow Def(insertar(x,l,i))$

ecuaciones

{Las listas empiezan en la posición 1}

$(x:l)[suc(0)] = x \; n > 0 \Rightarrow (x:l)[suc(n)] = l[n]$

{Insertar x en la posición 1 es poner x el primero de todos}

$insertar(x,l,suc(0)) = x:l$

{Insertar x después de la posición 1 es dejar el primero de la lista exactamente igual y seguir buscando el sitio de x}

$n > 0 \Rightarrow insertar(x,y:l,suc(n)) = y:insertar(x,l,n)$

{Borrar la posición 1 es quitar la cabeza de la lista}

$borrar(x:l,suc(0)) = l$

{Borrar después de la posición 1 es dejar la cabeza en su sitio y seguir buscando el que hay que borrar}

n > 0 ⇒ borrar(x:l,suc(n)) = x:borrar(l,n)

{Modificar el dato de la posición n es como si se quitase y luego se insertarse el dato nuevo}

n > 0 ⇒ modificar(x,l,n) = insertar(x,borrar(l,n),n)

{Para ver si un dato está en la lista se recorre y se compara con cada uno de los elementos de la lista, hasta que no queden más}

esta?(x,[]) = F

esta?(x,y:l) = (x eq y) ∨

esta?(x,l)

{Buscar la posición que ocupa un dato es mirar si está o no en la lista, y si está se va contando de uno en uno hasta llegar al dato}

esta?(x,l) = F ⇒ buscar(x,l) = 0

esta?(x,y:l) = T ∧ (x eq y) = T ⇒ buscar(x,y:l) = suc(0)

esta?(x,y:l)= T ∧ (x eq y)= F⇒buscar(x,y:l)= suc(buscar(x,l))

fespec

d) ARBOLES BINARIOS

espec ARBOLES_BINARIOS[ELEMENTO]

usa NATURALES2, BOOLEANOS

{NATURALES2 tiene operaciones para comparar numeros, como

max, min, s, etc.}

parametro formal

generos elemento

fparametro

generos a_bin farbol_binario}

A:> a_bin {Generadoras libres)

_._: a_bin elemento a_bin > a_bin

parcial raiz: a_bin > elemento

parcial izq: a_bin > a_bin

parcial der: a_bin > a_bin

vacio ?: a_bin > bool

parcial altura: a_bin > natural

var x: elemento a, i, d: a_bin

ecuaciones de definitud

vacia?(a) = F => Def(raiz[a]]

vacia?(a) = F=> Def(izq(a)]

vacia?(a) = F=> Def(der[a)]

vacia?(a) = F=> Def(altura[a)]

ecuaciones

raiz[i*x*d]=x

izq(i*x*d)=i

der[i*x.d) = d

vacio?(4) = T

vacioli*x*d]=F

altura[A*x*A)=0

vacio?(i] = F=> altura(i.x.A) = suc(altura[])

vacio?(d) = F => altura(A.x.d] = suc(altura[d))

(vacio?(i) = F) A (vacio?(d) = F) = alltura(i.x.d) =

suc[max|altura(i),altura[d))]

Fespec

e) ARBOLES BINARIOS+

espec ÁRBOLES_BINARIOS+[ELEMENTO]

usa ÁRBOLES_BINARIOS[ELEMENTO], LISTAS[ELEMENTO]

operaciones

preorden: a_bin -> lista {recorre en preorden}

inorden: a_bin -> lista {recorre en inorden}

postorden: a_bin -> lista {recorre en postorden}

var x: elemento i, d: a_bin

ecuaciones

preorden(4) = []

preorden(i.x.d) = [x] ++ preorden(i) ++ preorden(d)

inorden(A) = []

inorden(i.x.d) = inorden(i) ++ [x] ++ inorden(d)

postorden(4) = []

postorden(i·x·d) = postorden(i) ++ postorden(d) ++ [x]

fespec

f) ARBOLES DE BUSQUEDA+

espec ÁRBOLES_BÚSQUEDA+[ELEMENTO≤]

usa ÁRBOLES_BÚSQUEDA[ELEMENTO≤]

operaciones

borrar : elemento a_bin -> a_bin {quita un elemento}

{"borrar" total para simplificar las ecuaciones}

operaciones auxiliares

parcial máximo : a_bin -> elemento {busca el dato mayor}

var x, y: elemento; a, i, d: a_bin

ecuaciones de definitud

vacío?(a) = F = Def(máximo(a))

ecuaciones

máximo(i·x·A) = x

vacío?(d)=F= máximo(i.x.d) = maximo(d)

borrar(x,A) = 4

(y <x)=> borrar(y, i.x.d) = borrar(y, i).x.d

(y> x)=> borrar(y, i.x.d) = i.x.borrar(y, d)

(y=x)=> borrar(y, 4.x.d) = d

(y = x) A vacio?(i)=F => borrar(y, i.x.A) = i

(y = x) A vacio?(i)=F A vacio?(d)=F = borrar(y, i.x.d) =

borrar(máximo(i), i) · máximo(i) . d

Fespec

2. Resolución de dificultades encontradas

PARTE 1

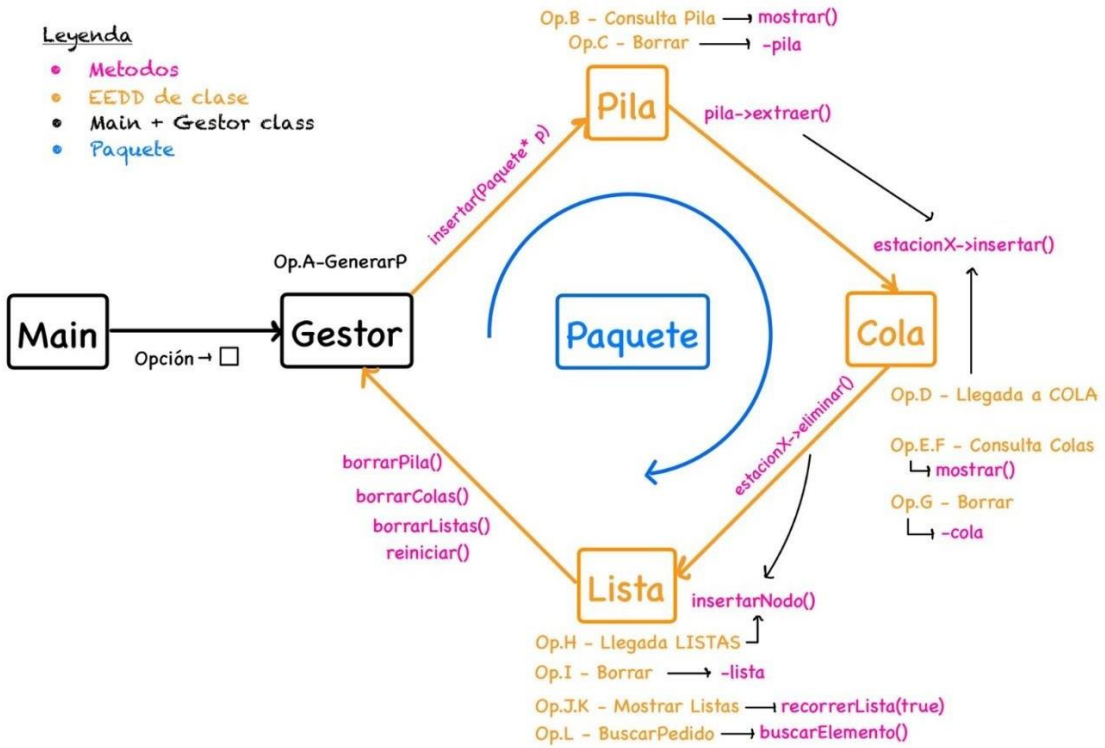
Los principales desafíos en el proceso de desarrollo del código se centraron en la adecuación de las estructuras de datos a los requisitos especificados en el enunciado de la práctica. Para abordar esta cuestión, se llevó a cabo una sesión de colaboración y se creó un diagrama que permitiera visualizar el flujo del programa. Las restricciones en cuanto a los rangos de las pilas, colas y listas se resolvieron de manera individualizada.

En el caso de las pilas, se implementó un condicional que limita el número máximo de paquetes en la pila a 49. Para las colas, se introdujeron dos atributos enteros estáticos que incrementan su valor solo de forma progresiva, generando identificadores únicos. Respecto a las listas, se diseñó una función denominada "isRepetido" que se encarga de verificar si un número de seguimiento se encuentra duplicado. En caso de duplicación, esta función genera un nuevo número dentro del rango y trata de reasignarlo. Es importante destacar que las restricciones en los ID de paquetes en la pila garantizan que nunca superemos la cantidad límite permitida en las listas.

La clase gestor presentó desafíos adicionales debido a su complejidad y al número de funciones que incorpora. Sin embargo, gracias a la planificación previa, se pudo anticipar y abordar de manera efectiva cualquier confusión relacionada con la pertenencia de métodos o atributos auxiliares, lo que agilizó la resolución de problemas.

El enfoque previamente establecido permitió una planificación más efectiva y una construcción más sólida del modelo del Gestor de Amazon, evitando potenciales problemas y facilitando una solución más eficiente.

Diagrama de las ideas iniciales:



PARTE 2

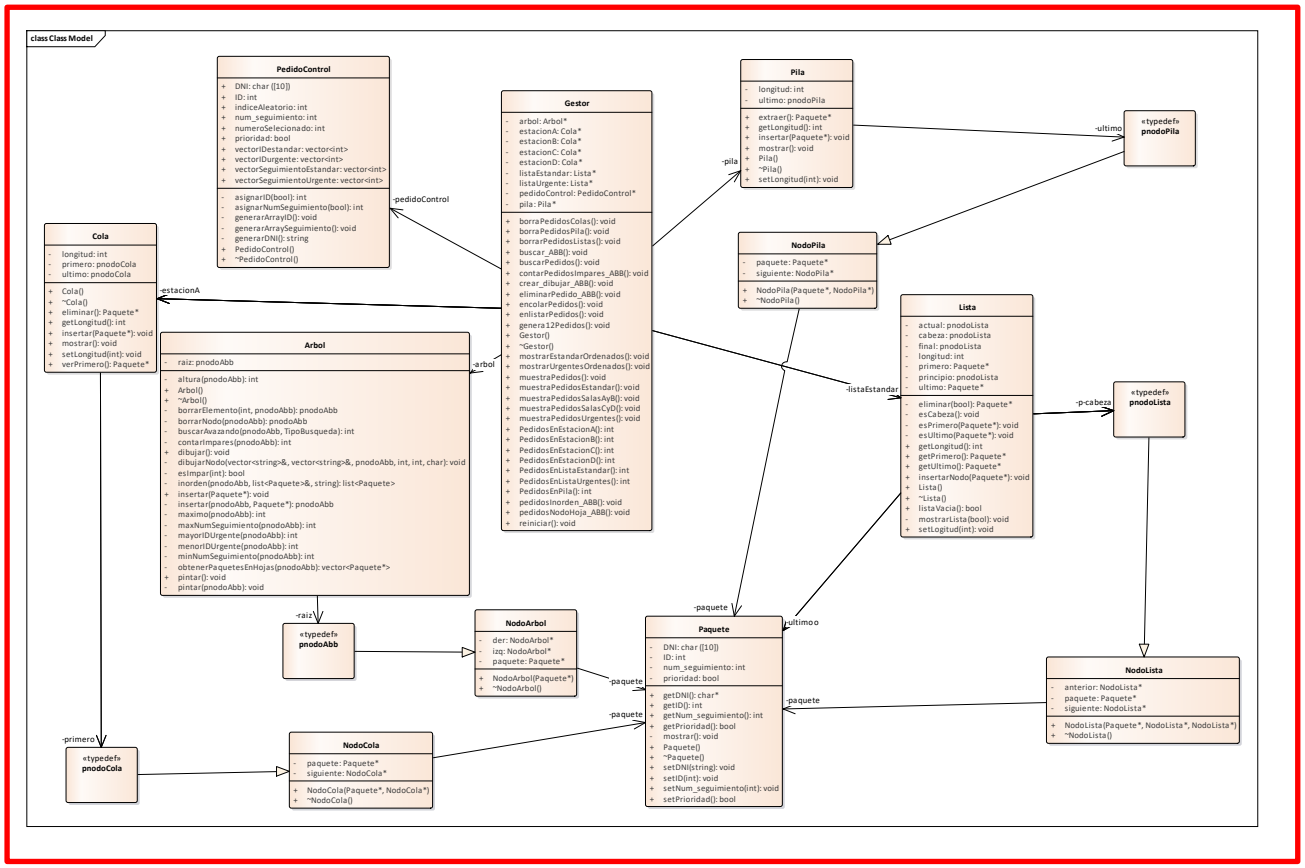
Durante la resolución de las dificultades encontradas en la defensa de la parte 1 del código, se han implementado diversas correcciones y mejoras para abordar los fallos identificados. En primer lugar, se ha trabajado en la adecuación de las funciones encargadas de generar números de seguimiento e identificadores, optando por utilizar un enfoque que evita la repetición de elementos. Para ello, se ha implementado la generación de un array y la extracción de elementos de manera que no se produzcan duplicados.

Otro aspecto abordado ha sido la corrección del formato de impresión por pantalla, mejorando la presentación de los resultados para una mejor legibilidad. Además, se han realizado ajustes en algunas estructuras esenciales del código para garantizar su correcto funcionamiento.

A pesar de que la mayor parte del código fue proporcionada por el profesor, se han realizado adaptaciones necesarias, como cambios en nombres de variables, para cumplir con los requisitos específicos de los datos solicitados.

Donde se experimentó un aumento en la complejidad fue en funciones como buscarAvanzado, eliminarElemento, y contarImpares. Sin embargo, mediante el seguimiento de los apuntes proporcionados y una comprensión detallada de la lógica subyacente, se logró modelar estas funciones con éxito. Cada función fue ajustada de manera que cumpliera con su funcionalidad específica, superando así los desafíos inherentes a su implementación.

3. Explicación de los métodos más destacados



A continuación se muestra una explicación de algunos métodos destacados de cada clase:

a) Paquete:

-**generarDNI()**: un método que genera una cadena de números aleatoria y les asigna una letra en función de la cadena numérica.

-**asignarID()**: este método utiliza dos variables estáticas para asignar un identificador único a los paquetes en función de la prioridad que tengan (urgente o estándar), que va de forma lineal del 1 al 49 y del 51 al 99.

-**generarNumSeguimiento(bool)**: también utiliza la prioridad para generar su número de seguimiento, y utiliza un vector de números en el que se almacenan los números de seguimiento para evitar tener repetidos pues cuando se genera un numero que coincide, vuelve a llamar recursivamente a la función. Los valores para los estándar están entre 1 y 499, y para los urgentes entre 501 y 999.

b) Pila/Cola:

-**mostrar()**: este método utiliza un puntero auxiliar para apuntar a cada paquete del TAD y mientras lo recorre va mostrando los datos de cada paquete.

c) Lista:

-**buscarElemento(char)**: este método realmente solo encuentra el paquete de la lista que está al principio de la lista estándar y al final de la lista urgente y muestra sus datos debido a que los datos entran a la lista ordenados de manera que el primero y el último son el de menor prioridad y mayor prioridad de sus respectivas listas (un requisito del programa).

-**recorrerLista(bool)**: permite con un puntero auxiliar recorrer los datos de la lista, ordenados en orden ascendente o descendente según se le indique con un booleano.

d) Gestor:

-**genera12Pedidos()**: crea nuevos paquetes y almacena su dirección de memoria en la pila. Tiene un límite de hasta 48 pedidos en la pila simultáneamente.

-**encolarPedidos()**: extrae los paquetes de la pila y los va introduciendo en las colas de manera que un paquete comprueba su prioridad que le indica si debe ir a las colas

estándar o urgentes, y existiendo dos colas de cada prioridad, va primero a una cola y luego a la otra, comprobando el tamaño de la cola para que alterne de 1 en 1 los paquetes que entran en cada una. Además detecta si la cola se llena y desecha los paquetes en ese caso mostrando por pantalla cual se ha desechado.

-**enlistarPedidos()**: toma los paquetes de las colas y los enlista en estándar y urgente según si proceden de las estaciones A y B o C y D. En este caso enlista B primero y A después para que el elemento con mayor ID de la lista quede en el principio y enlista C primero y D después para que el elemento con menor ID de la lista urgente quede el último. Este detalle nos ayuda a que en el método de lista "buscarElemento" no tengamos que recorrer toda la lista sino que conocemos la ubicación de los paquetes que necesitamos.

e) Arbol

- Eliminar elemento (Maximo, borrarElemento, borrarNodo) : La función maximo se encarga de encontrar y devolver el elemento máximo (o el nodo con el valor máximo) en un subárbol dado, lanzando una excepción si el árbol está vacío. La función borrarElemento busca y elimina un nodo con un valor específico (elementoAEliminar) en el árbol, llamando a la función auxiliar borrarNodo cuando se identifica el nodo a eliminar. La función borrarNodo maneja la eliminación real del nodo, considerando tres casos.

- **buscarAvazando**: El código implementa operaciones de búsqueda avanzada en un árbol binario de búsqueda (ABB) para encontrar valores mínimos y máximos de seguimiento, así como valores mínimos y máximos de ID urgente. Además, incluye funciones para verificar si un número es impar y contar la cantidad total de números impares en el árbol. Estas operaciones son utilizadas según el tipo de búsqueda especificado en la función buscarAvanzando.

- **obtenerPaquetesEnHojas**: La función obtenerPaquetesEnHojas recorre un árbol binario de búsqueda y devuelve un vector con los paquetes ubicados en las hojas del árbol. Realiza esto de manera eficiente, identificando nodos hoja y combinando los resultados de las ramas izquierda y derecha.

- **contarImpares**: Estas funciones trabajan en un árbol binario para contar eficientemente la cantidad total de números impares en los nodos del árbol.

4. Comportamiento del programa

El programa que hemos desarrollado tiene como objetivo simular un proceso lineal de gestión y envío de paquetes. Este proceso se inicia con un menú de opciones que es manejado por una clase llamada "Gestor". El Gestor se encarga de ejecutar las diferentes acciones disponibles en el menú, lo que incluye la generación y gestión de 12 paquetes.

Cuando se procede a la generación de estos paquetes, se almacenan en una pila, inicializando sus atributos básicos. Posponiendo la asignación del ID del paquete y el número de seguimiento, valores que se asignarán más adelante en el proceso. Los paquetes se mantienen temporalmente en la pila hasta que se asigna un ID y se determina su prioridad.

A continuación, los paquetes se distribuyen en cuatro colas, cada una destinada a un tipo de prioridad. Con el objetivo de simular estaciones de empaquetado, las estaciones A y B se utilizan para los pedidos estándar, mientras que las estaciones C y D están reservadas para los pedidos urgentes. Siguiendo las pautas de la práctica, se busca lograr un reparto equitativo de paquetes en estas cuatro colas.

Una vez que los paquetes han sido procesados en las estaciones de empaquetado, se les asigna un número de identificación único y se procede a enlistar. Estos paquetes se organizan en dos listas distintas, "listaEstandar" y "listaUrgente", según su prioridad. Esto permite una gestión más sencilla y eficaz de los paquetes y sus características.

El programa no solo se limita a estas operaciones básicas, sino que también ofrece otras funcionalidades. Puedes, por ejemplo, consultar y mostrar el contenido de estas estructuras, determinar su longitud y realizar operaciones de vaciado.

Para garantizar un funcionamiento adecuado del programa, hemos incorporado restricciones clave. Por ejemplo, la pila inicial no puede exceder los 48 paquetes generados, evitando una sobrecarga en el sistema. Asimismo, en las colas, se han establecido límites para los números de identificación de los paquetes. Para la lista de pedidos estándar, los IDs se encuentran en el rango de 1 a 49, mientras que para los pedidos urgentes, se sitúan en el rango de 501 a 999. Si se superan estos límites, el programa elimina paquetes durante el proceso de encolado. Además, se ha tenido especial cuidado en evitar la generación de números de ID repetidos, lo que se logra gracias a la definición de rangos únicos para pedidos estándar y urgentes.

El programa implementa una estructura de datos basada en un Árbol Binario de Búsqueda (ABB) para gestionar objetos de la clase Paquete. En su funcionamiento, se destaca la capacidad de insertar nuevos paquetes, que utiliza recursividad para encontrar la posición adecuada en el árbol según el número de seguimiento de los paquetes. Además, se proporcionan de antemano métodos de visualización gráfica del árbol, como pintar y dibujar, que permiten observar la disposición y jerarquía de los nodos en la consola.

El programa también incorpora funciones avanzadas de búsqueda, como buscarAvanzando, que ofrece flexibilidad al realizar búsquedas según distintos criterios especificados por el usuario. Además, se incluyen métodos de consulta en el árbol, como esImpar y contarImpares, que permiten verificar la paridad de los números de seguimiento y contar la cantidad total de números impares en el árbol, respectivamente.

Otro aspecto destacado es la función obtenerPaquetesEnHojas, que recopila eficientemente los paquetes ubicados en las hojas del árbol, devolviendo un vector de punteros a dichos paquetes. Asimismo, se han implementado métodos para eliminar nodos específicos del árbol, preservando la propiedad de árbol de búsqueda binario.

En conclusión, la clase Arbol proporciona una interfaz completa y robusta para la gestión de un árbol binario de búsqueda que almacena y manipula objetos Paquete. Este enfoque estructurado y eficiente facilita la manipulación y visualización de datos relacionados con paquetes, ofreciendo flexibilidad y versatilidad en las operaciones sobre el árbol.

5. Bibliografía

-Apuntes de clase

-GitHub: Let's build from here · GitHub