PL2 2023/24 Estructuras de datos

Jorge Barroso – 50359997W

Luciana Paola Diaz – Y6049376E

TAD'S CREADOS

- Pila [Mesa]
- Cola [Reserva]
- Lista [Pedido]
- Árbol binario búsqueda [Cadena]

Siendo Mesa, Reserva y Pedido clases del programa.

ESPECIFICACIÓN DE LOS TAD'S

Especificación del TAD Pila [Mesa]:

espec PILA[MESA]
usa BOOLEANOS, RESERVA, NATURALES
parámetro formal
    géneros mesa
fparámetro
géneros pila
operaciones
    Pila: -> pila {constructor de pila, equivalente a pilaVacia} (Generadora)
    apilar: pila mesa -> pila (Generadora)
    apilarEnOrden: pila mesa -> pila (Modificadora)
    Parcial desapilar: pila -> pila (Modificadora)
    vaciarPila: pila -> pila (Modificadora)
    generarMesas: pila (Modificadora)
    Parcial getMesaCima: pila-> mesa (Observadora)
    esVacia: pila -> boolean (Observadora)
    mostrarPilaMesas: pila
    buscarMesas: reserva natural -> puntero a array de ^mesa
var p: pila
ecuaciones de definitud
    esVacia(p) = F -> Def(desapilar(p))
    esVacia(p) = F -> Def(getMesaCima(p))
fespec

Especificación del TAD Cola[Reserva]:

    Parcial desencolar: cola -> cola (Modificadora)
    generarReservas: cola (Modificadora)
    Parcial inicio: cola -> reserva (Observadora)

Parcial fin: cola -> reserva (Observadora)
esVacia: cola -> boolean (Observadora)
Parcial mostrarCola: cola

var cola: c

ecuaciones de definitud

esVacia(c) = F -> Def(desencolar(c))
esVacia(c) = F -> Def(inicio(c))
esVacia(c) = F -> Def(fin(c))
esVacia(c) = F -> Def(mostrarCola(c))

Especificación del TAD Lista[Pedido] (Lista doblemente enlazada):

espec LISTA[PEDIDO]
usa BOOLEANOS, PILAS
parámetros formales
genéros pedido
fparámetro
genéros lista
operaciones

Lista: -> lista (Generadora)
extenderListaPorIzquierda: lista pedido -> lista (Generadora)
extenderListaPorDerecha: lista pedido -> lista (Modificadora)
Parcial eliminarPrimero: lista -> lista
(Modificadora)
Parcial eliminarUltimo: lista -> lista (Modificadora)
Parcial completarSiguientesPedidos: pila lista -> lista (Modificadora)

esVacia: lista -> boolean (Observadora)
Parcial mostrarDatosLista: lista
Parcial getPrimero: lista -> NodoLista* (Observadora)

extenderListaPorCategoria: pedido -> Lista (Modificadora)
Parcial sacarSiguientesPedidos: Pila -> Lista (Modificadora)

var l:Lista, p:Pedido , pila:Pila

ecuaciones de <u>definitud</u>

esVacia(l)=F -> Def(extenderListaPorDerecha(l,p))
esVacia(l)=F -> Def(eliminarPrimero (l))
esVacia(l)=F -> Def(eliminarUltimo (l))
esVacia(l)=F -> Def(completarSiguientesPedidos (pila,l))
esVacia(l)=F -> Def(mostrarDatosLista (l))
esVacia(l)=F -> Def(getPrimero (l))

esVacia(l)=F -> Def(sacarSiguientesPedidos(pila))

fespec

Especificación del TAD arboles_binarios[Cadena]:

espec ARBOL_BINARIO [CADENA]
usa NATURALES2, BOOLEANOS (NATURALES2 contiene operaciones para comparar números como: máx., min, <, >, etc.)
parámetro formal
Géneros CADENA
fparametro
Géneros a_bin (árbol binario)
Operaciones
a_bin: -> a_bin (Generadoras)
fespec

Especificación del TAD arboles_búsqueda[Cadena =<]:

espec ARBOLES_BUSQUEDA [CADENA =<]
usa ARBOLES_BINARIOS [CADENA], PEDIDO, LISTA
Parámetro formal
Géneros CADENA
Operaciones
todas son operaciones: cadena cadena -> bool
(<,=<,>,=>,=)
Var: X,Y: cadena
fparametro
Géneros ABB (árbol binario de búsqueda)
Operaciones
Insertar: cadena pedido-> ABB (insertar ordenadamente) (modificadora)
VerEnOrden: bool bool -> void (Observadora)
BuscarListaPedidosPorNombre: cadena -> Lista
SumarCategorias: (Observadora)
fespec

DESCRIPCIÓN DE LAS DIFICULTADES ENCONTRADAS Y SOLUCIÓN ADOPTADA

A lo largo de la práctica hemos tenido una serie de dificultades con métodos, tipos de datos y las especificaciones de los TAD's con métodos añadidos:

1. Uno de los puntos más difíciles de la práctica es el uso de la memoria dinámica al modificar las listas o arboles binarios de búsqueda. En específico en las funciones implementadas en lista como extenderPorCategoria() o sacarSiguientesPedidos() en las cuales era necesario manejar bien los punteros al insertar los pedidos de manera que se mantuviesen enlazados correctamente y en el orden que requería

la práctica. Esto derivó en varios problemas de conectividad entre los NodosLista al llegar a la función de finalizar la mitad de los pedidos de cada categoría, ya que al no estar bien enlazados daba errores al intentar sacar los pedidos de la lista y meterlos en otra para ser insertados en el árbol binario de búsqueda. Esto retrasó el flujo de la práctica y se tuvieron que rehacer las funciones de nuevo para poder entender el problema y solucionarlo, reescribiendo las funciones de nuevo con un objetivo diferente y usando diagramas para entender a fondo el código.

2. Otro punto importante es un problema de referencia circular que tuvimos entre las clases ABB, Lista y NodoABB. Estas clases se referenciaban entre ellas derivando a un problema mayor de referencia circular, por lo que este error retrasó aún más el flujo de trabajo de la práctica. La consecuencia de esto era que no se podía compilar bien, ni probar, las funciones implementadas en la clase ABB ya que para poder visualizar el funcionamiento del árbol se tenía que visualizar como se iban añadiendo los pedidos a los NodosABB de los clientes. Para solucionarlo se modificó la función previamente encargada de sacar la mitad de los pedidos de la lista de pedidos e insertarlos en el árbol, para que en lugar de insertarlos directamente (lo que requería de acceso a ABB.h) se devolviera una lista con los pedidos extraídos a la clase Gestor y que esta se encargará de insertarlos en el ABB, evitando así la referencia circular.

DISEÑO DE LA RELACIÓN ENTRE LAS CLASES DE LOS TAD IMPLEMENTADOS

En nuestro programa hemos implementado las siguientes clases:

- Pila
- NodoPila
- Cola
- NodoCola
- Lista (Doblemente enlazada)
- NodoLista
- Pedido
- Reserva
- Mesa
- ABB
- NodoABB
- Gestor
- main

Para empezar, las cuatro colecciones implementadas (Pila, Cola, Lista y ABB) están formadas por nodos (NodoPila, NodoCola, NodoLista y NodoABB respectivamente) que guardan un valor puntero a otra clase (Mesa, Reserva, Pedido y par nombre-Lista respectivamente) y un puntero al siguiente nodo llamado “siguiente”. Para entenderlo mejor, estos son los atributos de NodoPila:

```
Mesa* pmesa;
```

NodoPila *siguiente;
friend class Pila;

Notese que usa friend class para permitir usar estos atributos privados a una clase especificada, en este caso Pila. Cola y Lista realizan lo mismo, como se puede ver en el código.

Además, la clase NodoLista como la lista es doblemente enlazada guarda también un puntero NodoLista al nodo anterior.

De esta forma cada clase colección maneja un tipo distinto de datos y está relacionada con sus mismas clases nodos siendo gráficamente sus relaciones:

Pila->NodoPila->Mesa
Cola->NodoCola->Reserva
Lista->NodoLista->Pedido

ABB->NodoABB->Cadena,Lista

Finalmente, en la clase gestor se crean todas las instancias necesarias de las colecciones (3 colas, una pila, una lista y un árbol de búsqueda) y se van modificando a lo largo del funcionamiento del programa.

EXPLICACIÓN DEL COMPORTAMIENTO DEL PROGRAMA Y DE LOS MÉTODOS MÁS DESTACADOS

Antes de proceder a la explicación, debe saberse que todo el código del programa está documentado debidamente para una mayor claridad y legibilidad del programa, por lo que si esta breve explicación no fuese suficiente para entender el programa, se puede leer directamente el código fuente documentado.

La ejecución del programa comienza en el archivo main.cpp, dónde anteriormente se inicializaban las variables, pero ahora solo se ejecutan las distintas operaciones que ofrece el menú. Dichas variables ahora se inicializan en la clase Gestor por lo que no se necesitaría pasar como referencia las distintas estructuras de datos.

Dentro del main() se entra en un bucle en el cual se mostrará el menú principal al usuario con las posibles opciones a realizar, pedirá la elección de una al usuario, la validará y llevará a cabo, y se reiniciará el bucle a menos que se escoja la opción 19. Salir, en cuyo caso el bucle termina y finaliza el programa.

Dependiendo de la opción que se lleve a cabo se llamará a un método de la clase Gestor, que llevará a cabo dicha opción. Para cada opción del menú hay un método correspondiente en la clase Gestor. Además, en el caso de elegir la opción 0 (Generar la cola de reservas) se creará aleatoriamente el número de reservas a gestionar desde un mínimo de 12 a un máximo de 50 reservas. Para las opciones 6, 7 y 8 se llamará al método correspondiente de Gestor y después de este se llamará al método mostrarDatos de Gestor que mostrará todos los datos correspondientes tras las simulaciones, además en la opción 8 se llamará a mostrarAbbPedidos de Gestor que mostrará todos los clientes con sus respectivos pedidos ya gestionados.

Se procederá a explicar los métodos nuevos más relevantes de la clase Gestor, obviando la explicación de los métodos anteriores ya que se ha mencionado en la memoria anterior.

La **opción 9** corresponde con una nueva función que permite añadir hasta un máximo de 10 pedidos en una misma pasada, tomando los datos en cada pedido con la función cogerDatosPedido del nombre del cliente, el número de personas, la preferencia del menú, etc. Además, se llama a otra función que se encarga de validar el entero que sea insertado en la terminal para evitar posibles errores o salidas de la terminal al insertar cualquier otro carácter que no sea un entero. Tras finalizar la función de cogerDatosPedido, intenta insertar dichos pedidos con las mesas libres del momento, en caso de que un pedido no encuentre la función finaliza añadiendo los pedidos anteriores que si hayan conseguido mesa y los posteriores los descarta.

Las 3 siguientes opciones del menú principal corresponden a las funciones para mostrar la cola de reservas, la lista de pedidos y la cola de reservas pendientes.

La **opción 13** añade un pedido directamente al árbol binario de búsqueda. Tiene la misma ejecución que la opción 9 llamando a la función cogerDatosPedidos y si encuentra mesas lo añade al árbol binario de búsqueda en vez de a la lista de pedidos, y se toma como un pedido ya finalizado por lo que las mesas vuelven a estar libres. Además, al insertar un pedido al árbol binario de búsqueda, se inserta usando una estructura de inserción recursiva que los inserta de forma que el árbol este ordenado alfabéticamente por nombre de A a Z. De esta forma el ABB está siempre ordenado lo que permite buscar los pedidos por nombre de cliente más eficientemente.

Las siguientes 5 opciones del menú principal corresponden a las funciones para mostrar los datos del árbol binario de pedidos; Los nombres de los clientes con al menos un pedido gestionado (que es lo mismo que ver todos los nombres almacenados en el árbol binario de búsqueda); Los pedidos realizados por un cliente en específico, en el que se pide por pantalla el nombre del cliente que se quiere mostrar; El número de pedidos realizados por categorías y por último mostrar los clientes que han pedido un menú vegano. Todas estas últimas están implementadas en ABB.cpp y utilizan una función principal que llama a otra función auxiliar con estructura recursiva que explora el árbol de búsqueda teniendo en cuenta que esta ordenado por nombre de cliente alfabéticamente y o bien muestra su contenido con algunas restricciones (como solo vegano para la opción de solo menús veganos) o bien cuenta los números de cada categoría de pedidos.

BIBLIOGRAFÍA

- Transparencias de clase: Profesora: Mª José Domínguez Alda, Departamento de Ciencias de la computación, Universidad de Alcalá, Alcalá de Henares, España; **uah.blackboard.com/2023-24: ESTRUCTURAS DE DATOS (2A_G781) / Teoría/**
    - **TEMA 1-INTRODUCCIÓN TIPOS ABSTRACTOS DE DATOS.**
    - **TEMA 2-TAD PILA** y **TEMA 3-TAD COLAS**
    - **TEMA 4-TAD LISTAS** y **TEMA 4B-TAD LISTA+**
    - **TEMA 5-ARBOLES BINARIOS Y DE BUSQUEDA**
- **Youtube/Neso Academy/ Introduction to Doubly Linked List:** https://www.youtube.com/watch?v=e9NG_a6Z0mg
- Other **Youtube/Neso Academy** double linked list videos.