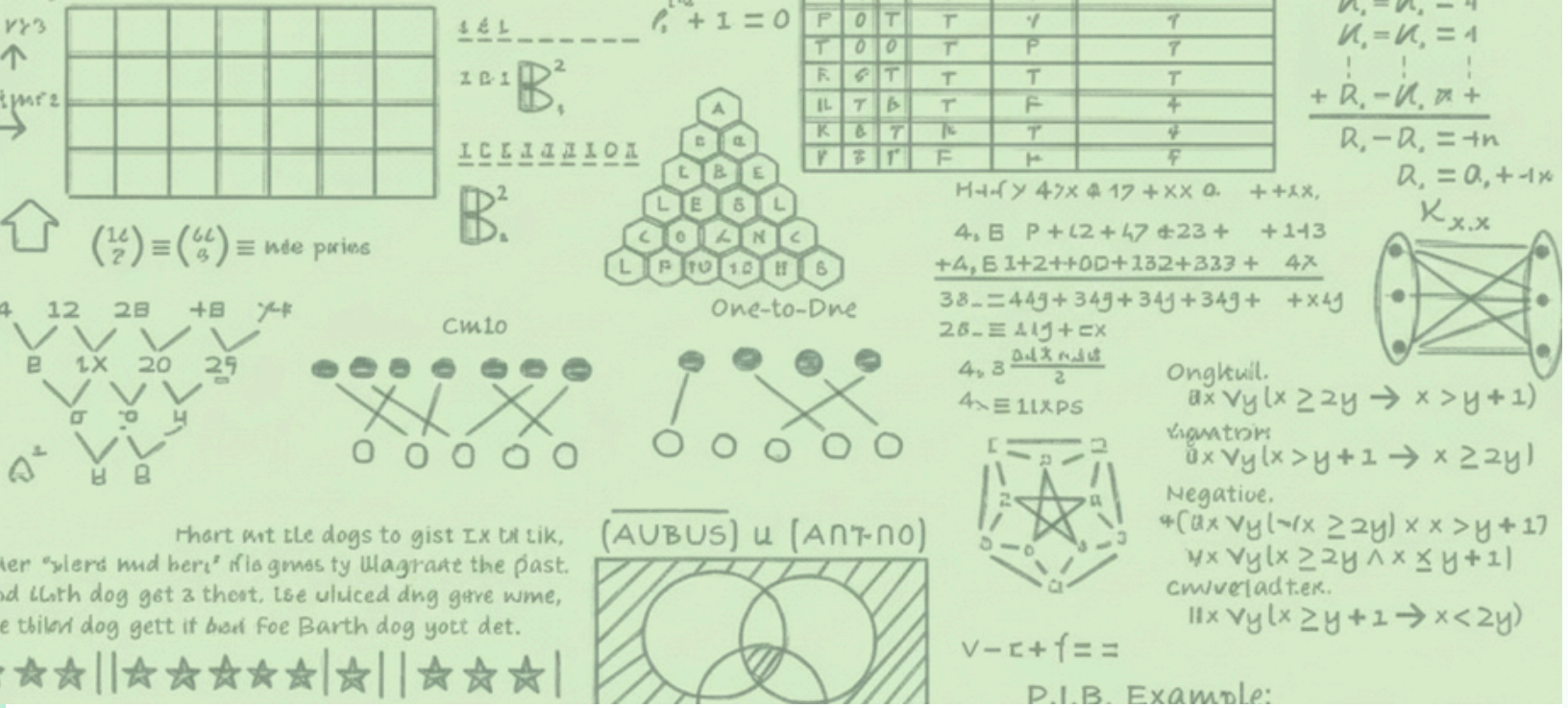


Proyecto Final

Estructuras de datos

Presentado por

Fernando Núñez
Ricardo Baranda
David
Mariñelarena

Universidad Tecmilenio

Introducción

Como se había presentado anteriormente, en el restaurante Dolce Alba, se implementó un sistema de gestión del mismo, por medio de listas para los platillos, pilas para acomodar insumos y colas de prioridad para administrar las filas de clientes.
Si bien, el sistema implementado fue clave para impulsar el desarrollo del restaurante, ese mismo desarrollo y crecimiento requiere de un sistema más preparado, con ello en el presente reporte se muestra un sistema más avanzado, preparado con árboles binarios especializados en gestión de empleados por departamento, recursividad y algoritmos, tablas hash y clases hashmap, métodos de ordenamiento y búsqueda además de grafos, todo esto con la finalidad de tener un mejor control y gestión del restaurante.

Cola de prioridades

Una cola de prioridades garantiza que en la prioridad definida (1 con reservación y 2 sin reservación) se atienda a un cliente que tiene reservación sin errores antes que a un cliente que no la tiene.
Las clases involucradas en la creacion de la cola de prioridades fueron “Node, PriorityQueue y colas:

```
class Node {
    String name; // nombre del cliente
    int priority; // 1 = con reservación (mas urgente), 2 = sin reservacion
    Node next;

    public Node(String name, int priority) {
        this.name = name;
        this.priority = priority;
        this.next = null;
    }
}


 // Cola con prioridad para clientes
 class PriorityQueue {
     private Node front; // frente de la cola (mayor prioridad al frente)

     public PriorityQueue() {
         front = null;
     }

  // Insertar cliente en funcion de su prioridad (1 mejor que 2)
  public void enqueue(String name, int priority) {
     Node newNode = new Node(name, priority);

     if (front == null || priority < front.priority) {
         newNode.next = front;
         front = newNode;
     } else {
         Node temp = front;
         while (temp.next != null && temp.next.priority <= priority) {
             temp = temp.next;
         }
         newNode.next = temp.next;
         temp.next = newNode;
     }
     System.out.println(Colores.CIAN + "Agregado: " + Colores.BLANCO + etiqueta(newNode));
  }
```

Las características del diseño de la cola es que tiene una lista enlazada propia (java.util.PriorityQueue), En cuanto a su prioridad: 1 = significa tener reservación (prioridad más alta) y 2 = significa que no tiene reservación.
Además de ser de carácter Invariante ya que la cola se mantiene ordenada por prioridad (todos los 1 van delante de los 2).

Cola de prioridades

Y en cuanto a su funcionamiento interno está conformado por:

- enqueue(String name, int priority): Sirve para Insertar en la posición correcta según prioridad.
- Si la cola está vacía o llega un 1 y el frente es 2, el nuevo nodo se vuelve el del frente.
- Pero si lo último no fuera el caso recorrería hasta encontrar el primer nodo con prioridad mayor en la cola y encadenarlo.
- dequeue(): Atiende (elimina) el frente.
- removeByName(String name): Elimina un cliente por nombre en cualquier posición.
- peek(): Muestra el siguiente a atender.
- display(): Recorre y muestra la cola completa.

```
// Elimina por nombre (en cualquier posicion)
public void removeByName(String name) {
    if (front == null) {
        System.out.println(Colores.ROJO + "La cola está vacia" + Colores.RESET);
        return;
    }
    if (front.name.equalsIgnoreCase(name)) {
        System.out.println(Colores.ROJO + "Eliminado: " + Colores.BLANCO + etiqueta(front));
        front = front.next;
        return;
    }
```

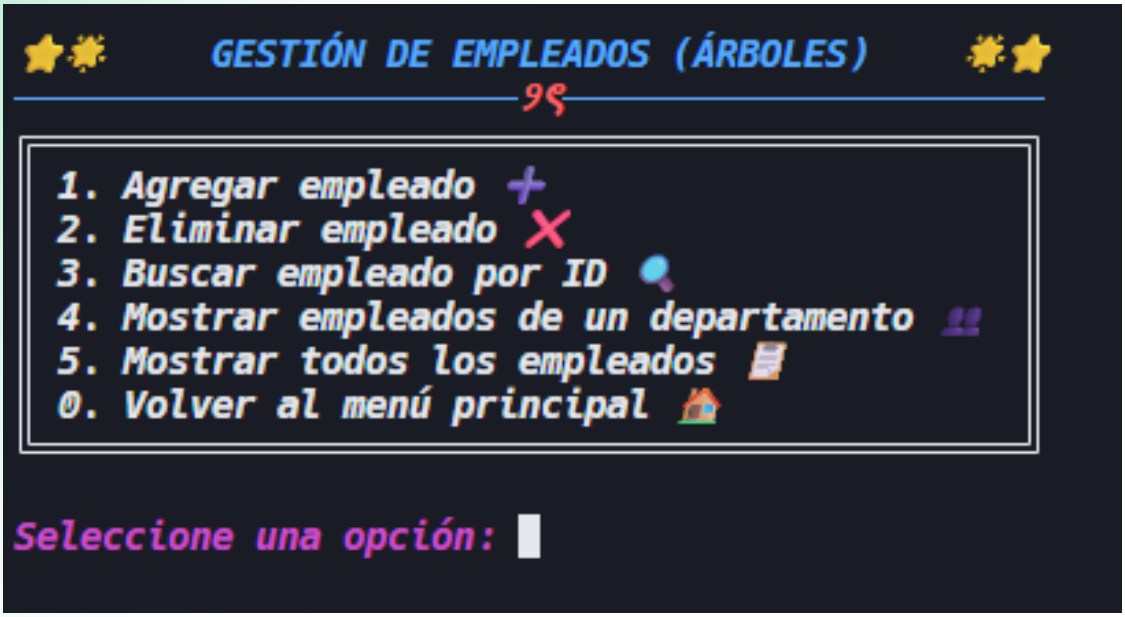
```
s Colas {
public static void gestionarClientes() {
    Scanner sc = new Scanner(System.in).useLocale(Locale.US);
    PriorityQueue cola = new PriorityQueue();

    while (true) {
        System.out.println(Colores.AZUL + "\n⭐🌟      GESTIÓN DE CLIENTES (COLAS)     🌟⭐" + Colores.RESET);
        System.out.println(Colores.AZUL + "————————————————————" + Colores.ROJO + "୨୧" + Colores.AZUL
                + "————————————————————" + Colores.RESET);
        System.out.println(Colores.BLANCO + "╔═══════════════════════════════════════════╗" + Colores.RESET);
        System.out.println(Colores.BLANCO + "║ 1. Agregar cliente                        ║");
        System.out.println("║ 2. Mostrar cliente al frente de la cola   ║");
        System.out.println("║ 3. Atender cliente al frente de la cola   ║");
        System.out.println("║ 4. Eliminar cliente por nombre            ║");
        System.out.println("║ 5. Ver toda la cola                       ║");
        System.out.println("║ 0. Salir                                  ║" + Colores.RESET);
        System.out.println(Colores.BLANCO + "╚═══════════════════════════════════════════╝" + Colores.RESET);
        System.out.print(Colores.MORADO + "\nSelecciones una opción: " + Colores.RESET + Colores.BLANCO);

        String op = sc.nextLine().trim();
```

Árboles Binarios

Para la implementar los árboles binarios se pensó en su uso para la gestión de empleados, como podría ser la agregación de un nuevo elemento al equipo de Dolce Alba, incluso una baja en dado caso, encontrar el empleado y su puesto de acuerdo a su identificador único (ID), mostrar a todos los empleados de un departamento y listar a todos los empleados.



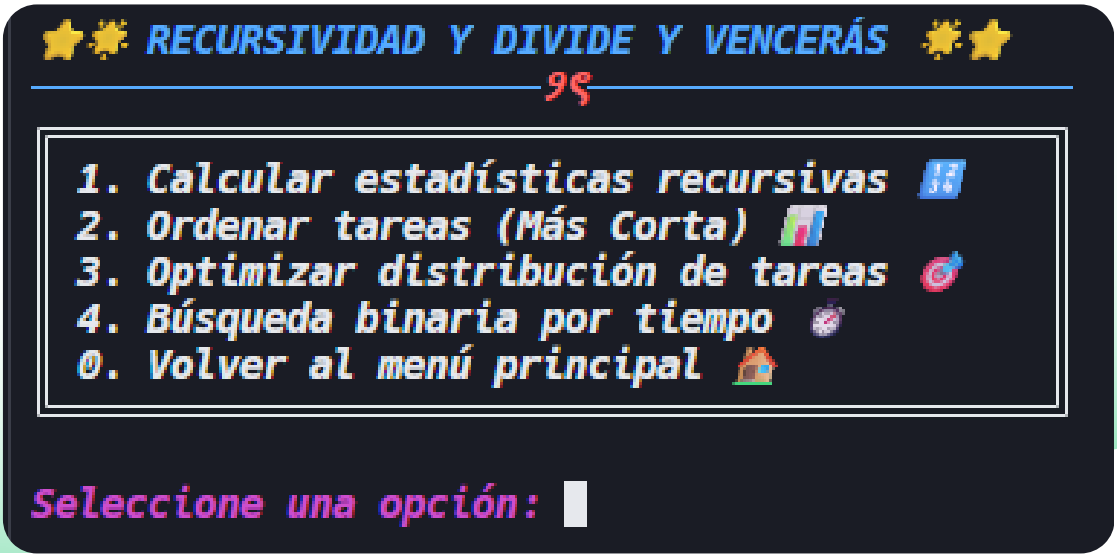
Cada opción definida con un método propio, y la clase de empleados funcionando como un nodo.

```
static class NodoEmpleado {
    int idEmpleado;
    String nombre;
    String puesto;
    NodoEmpleado izquierdo;
    NodoEmpleado derecho;

    public NodoEmpleado(int idEmpleado, String nombre, String puesto) {
        this.idEmpleado = idEmpleado;
        this.nombre = nombre;
        this.puesto = puesto;
    }
```

Y cada uno incluye dos apuntadores para de esta manera recorrer el árbol y encontrar los empleados de forma rápida.

Recursividad y algoritmos “divide y vencerás”



El módulo de Recursividad y Divide y Vencerás del sistema Dolce Alba implementa algoritmos avanzados para la optimización y análisis de tareas del restaurante. Esta sección utiliza técnicas de programación recursiva y el paradigma divide y vencerás para resolver problemas complejos de manera eficiente.

Funcionalidades del Sistema

- **Estadísticas Recursivas:** Tiempo total, conteo por prioridad, tarea de mayor duración
- **Ordenamiento Inteligente:** Merge sort para organizar tareas
- **Optimización:** Distribución automática por departamento con priorización
- **Búsqueda Eficiente:** Búsqueda binaria en listas ordenadas

```java
// Estructura para representar una tarea del restaurante
static class Tarea {
    String nombre;
    int tiempoEstimado; // en minutos
    int prioridad; // 1 = alta, 2 = media, 3 = baja
    String departamento;
    LocalDate fechaEntrega;
```

Clase Tarea

Propósito: Representa una tarea del restaurante con información sobre tiempo estimado, prioridad, departamento responsable y fecha de entrega.

Lista de Tareas (ArrayList)

Se utiliza List<Tarea> tareas para almacenar todas las tareas del restaurante, permitiendo operaciones eficientes de acceso y manipulación.

```java
    // Calcular tiempo total estimado usando recursividad
    public int calcularTiempoTotalRecursivo(List<Tarea> listaTareas, int indice) {
        // Caso base: si llegamos al final de la lista
        if (indice >= listaTareas.size()) {
            return 0;
        }

        // Caso recursivo: tiempo actual + tiempo del resto de la lista
        return listaTareas.get(indice).tiempoEstimado +
               calcularTiempoTotalRecursivo(listaTareas, indice + 1);
    }
```

Algoritmos Implementados

A. Algoritmos Recursivos

1. Cálculo de Tiempo Total Recursivo


```java
    // Calcular tiempo total estimado usando recursividad
    public int calcularTiempoTotalRecursivo(List<Tarea> listaTareas, int indice) {
        // Caso base: si llegamos al final de la lista
        if (indice >= listaTareas.size()) {
            return 0;
        }

        // Caso recursivo: tiempo actual + tiempo del resto de la lista
        return listaTareas.get(indice).tiempoEstimado +
               calcularTiempoTotalRecursivo(listaTareas, indice + 1);
    }
```


Merge Sort para Ordenamiento de Tareas

Propósito: Ordena las tareas por tiempo estimado utilizando el algoritmo merge sort.

Algoritmos Implementados

B. Algoritmos Divide y Vencerás

Funcionamiento del Sistema Recursivo

1.- Calcular estadísticas recursivas

```
=== ESTADÍSTICAS CALCULADAS RECURSIVAMENTE ===
🕐 Tiempo total estimado: 325 minutos (5.42 horas)
📌 Tareas de prioridad Alta: 3
📌 Tareas de prioridad Media: 3
📌 Tareas de prioridad Baja: 2
🏆 Tarea de mayor duración: Capacitar meseros (90 min)
```

2.- Ordenar tareas (Más Corta)

```
=== TAREAS ORDENADAS POR TIEMPO (MERGE SORT) ===
  • Limpiar mesas          |  15 min | Prioridad 2 | Sala | 2025-09-28
  • Revisar reservaciones  |  20 min | Prioridad 1 | Administración | 2025-09-28
  • Actualizar carta vinos |  25 min | Prioridad 3 | Sala | 2025-10-05
  • Inventario ingredientes |  30 min | Prioridad 1 | Cocina | 2025-09-30
  • Preparar postres       |  40 min | Prioridad 2 | Cocina | 2025-09-29
  • Preparar mise en place |  45 min | Prioridad 1 | Cocina | 2025-09-28
  • Preparar pan           |  60 min | Prioridad 2 | Cocina | 2025-09-29
  • Capacitar meseros      |  90 min | Prioridad 3 | Administración | 2025-10-12
```

3.- Optimizar distribución de tareas

```
=== DISTRIBUCIÓN OPTIMIZADA DE TAREAS ===

--- Sala ---
  • Limpiar mesas          |  15 min | Prioridad 2 | Sala | 2025-09-28
  • Actualizar carta vinos |  25 min | Prioridad 3 | Sala | 2025-10-05
  ⏱Tiempo total del departamento: 40 minutos

--- Cocina ---
  • Inventario ingredientes |  30 min | Prioridad 1 | Cocina | 2025-09-30
  • Preparar mise en place |  45 min | Prioridad 1 | Cocina | 2025-09-28
  • Preparar postres       |  40 min | Prioridad 2 | Cocina | 2025-09-29
  • Preparar pan           |  60 min | Prioridad 2 | Cocina | 2025-09-29
  ⏱Tiempo total del departamento: 175 minutos

--- Administración ---
  • Revisar reservaciones  |  20 min | Prioridad 1 | Administración | 2025-09-28
  • Capacitar meseros      |  90 min | Prioridad 3 | Administración | 2025-10-12
  ⏱Tiempo total del departamento: 110 minutos
```

4.- Búsqueda binaria por tiempo

```
Ingrese el tiempo en minutos a buscar: 90
```

```
✅ Tarea encontrada:
  • Capacitar meseros      |  90 min | Prioridad 3 | Administración | 2025-10-12
```

Tablas Hash y Clase HashMap

En la situación de tener varios empleados repartidos en diferentes departamentos una tabla hash sirve para poder encontrar a un empleado mediante su ID sin necesidad de saber su departamento.

La tabla hash utiliza un Índice global de empleados por búsqueda de orden de 1, el tiempo de búsqueda es constante y no crece cuando aumenta la cantidad de elementos apoyándose a su vez en los árboles binarios de búsqueda (ABB).

Las clases involucradas en su funcionamiento son las siguientes:

GestionEmpleados, ArbolBinarioBusqueda, NodoEmpleado

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.HashMap;
import java.util.Map;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Collections; // Needed for sorting
import java.util.Arrays; // Needed for Arrays.fill() in GrafoMesas

// Gestión de empleados por departamento
class GestionEmpleados {
    private Map<String, ArbolBinarioBusqueda> arbolesPorDepartamento = new HashMap<>();

    public void agregarEmpleado(String departamento, int idEmpleado, String nombre, String puesto) {
        ArbolBinarioBusqueda arbol = arbolesPorDepartamento.computeIfAbsent(departamento, k -> new ArbolBinarioBusqueda());
        arbol.insertar(idEmpleado, nombre, puesto);
        Main.limpiarPantalla();
        System.out.println(Colores.VERDE + "\n✅ Empleado agregado exitosamente." + Colores.RESET);
    }

    public NodoEmpleado buscarEmpleado(String departamento, int idEmpleado) {
        ArbolBinarioBusqueda arbol = arbolesPorDepartamento.get(departamento);
        return (arbol != null) ? arbol.buscar(idEmpleado) : null;
    }

    public void mostrarEmpleadosDepartamento(String departamento) {
        ArbolBinarioBusqueda arbol = arbolesPorDepartamento.get(departamento);
        if (arbol != null) {
            System.out.println(Colores.AZUL + "\n=== Organigrama del Departamento: " + Colores.AMARILLO +
                    departamento + Colores.AZUL + " ===" + Colores.RESET);
            arbol.recorridoEnOrden();
            System.out.println(Colores.AZUL + "=========================================" + Colores.RESET);
        } else {
            System.out.println(Colores.ROJO + "\nEl departamento '" + departamento +
                    "' no tiene empleados registrados." + Colores.RESET);
        }
    }
```

```
public NodoEmpleado buscarEmpleado(String departamento, int idEmpleado) {
    ArbolBinarioBusqueda arbol = arbolesPorDepartamento.get(departamento);
    return (arbol != null) ? arbol.buscar(idEmpleado) : null;
}
```

Tablas Hash y Clase HashMap

Respecto a su funcionamiento y una ventaja de este es que no es necesario conocer el departamento para buscar a un empleado; el HashMap lo resuelve al instante y luego el árbol binario de búsqueda (ABB) confirma el nodo.
Estructura base:

- Map<String, ArbolBinarioBusqueda> arbolesPorDepartamento = Significa que cada departamento mantiene un árbol binario de búsqueda (ABB) por idEmpleado.

- Índice HashMap global está conformado por: Map<Integer, String> indiceDepartamentoPorId, id y departamento.

- Se puede invocar a agregarEmpleado y todo lo que esto involucra (depto, id, nombre, puesto)

- Se puede Insertar al empleado en el arbol binario de busqueda (ABB) del departamento.

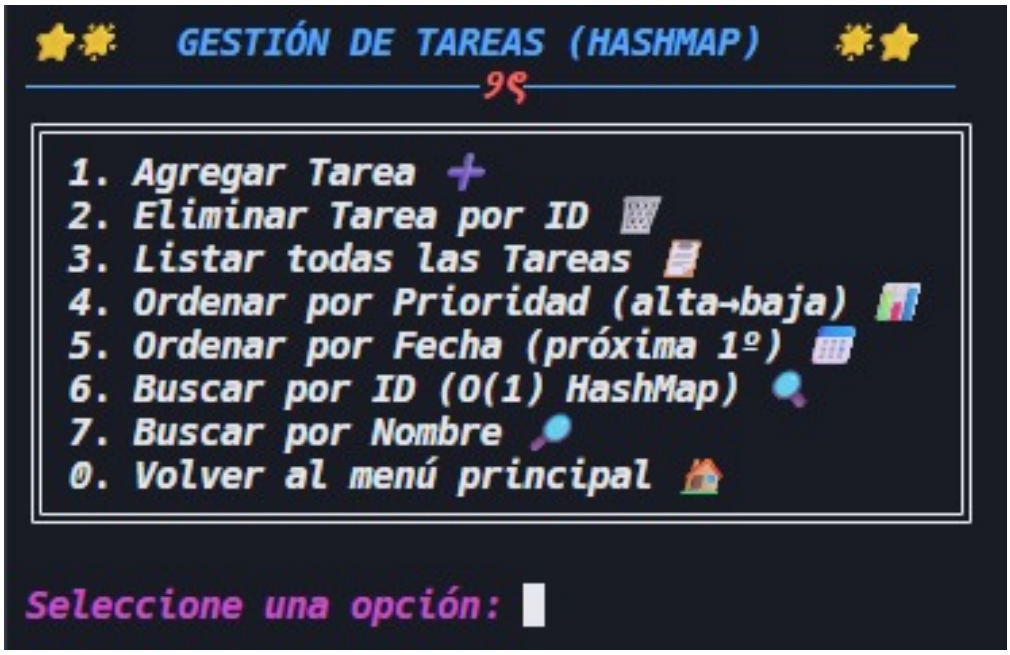
- Actualizado de HashMap global por la funcion de : indiceDepartamentoPorId.put(id, depto).

- Se puede buscar a un empleado por Id mediante *int id*

- Se puede Consultar la búsqueda en orden de 1 en el departamento con el metodo indiceDepartamentoPorId.get(id).

- Y por último usa el arbol de busqueda binario (ABB) de dicho departamento para ubicar el nodo exacto por id.

Tablas Hash y Clase HashMap




```java
// Clase para la gestión de tareas usando HashMap
class GestorTareas {
    private Map<Integer, RecursividadDivideVenceras.Tarea> tareasMap = new HashMap<>();
    private static final DateTimeFormatter FMT = DateTimeFormatter.ISO_LOCAL_DATE;
    private int nextId = 1; // Para asignar IDs únicos

    public GestorTareas() {
        inicializarTareas();
    }

    // Inicializar tareas predefinidas del restaurante
    private void inicializarTareas() {
        LocalDate hoy = LocalDate.now();
        agregarTareaInterna("Preparar mise en place", 45, 1, "Cocina", hoy);
        agregarTareaInterna("Limpiar mesas", 15, 2, "Sala", hoy);
        agregarTareaInterna("Inventario ingredientes", 30, 1, "Cocina", hoy.plusDays(2));
        agregarTareaInterna("Preparar pan", 60, 2, "Cocina", hoy.plusDays(1));
        agregarTareaInterna("Actualizar carta vinos", 25, 3, "Sala", hoy.plusDays(7));
        agregarTareaInterna("Revisar reservaciones", 20, 1, "Administración", hoy);
        agregarTareaInterna("Capacitar meseros", 90, 3, "Administración", hoy.plusDays(14));
        agregarTareaInterna("Preparar postres", 40, 2, "Cocina", hoy.plusDays(1));
    }
```

Métodos de ordenamiento y búsqueda

En una lista de tareas de un restaurante como la limpieza, preparación de alimentos, compras, mantenimiento etc. Los metodos de ordenamiento y busqueda sirven para dar prioridad a lo mas urgente, ver que tarea vence primero y encontrar rapidamente un elemento o tarea especifica.

Algunas de las clases y métodos forman parte del módulo de tareas de recursividad divide y vencerás por que utilizan las clases GestorTareas

```
// Clase para la gestión de tareas usando HashMap
class GestorTareas {
    private Map<Integer, RecursividadDivideVenceras.Tarea> tareasMap = new HashMap<>();
    private static final DateTimeFormatter FMT = DateTimeFormatter.ISO_LOCAL_DATE;
    private int nextId = 1; // Para asignar IDs únicos

    public GestorTareas() {
        inicializarTareas();
    }

// Inicializar tareas predefinidas del restaurante
private void inicializarTareas() {
    LocalDate hoy = LocalDate.now();
    agregarTareaInterna("Preparar mise en place", 45, 1, "Cocina", hoy);
    agregarTareaInterna("Limpiar mesas", 15, 2, "Sala", hoy);
    agregarTareaInterna("Inventario ingredientes", 30, 1, "Cocina", hoy.plusDays(2));
    agregarTareaInterna("Preparar pan", 60, 2, "Cocina", hoy.plusDays(1));
    agregarTareaInterna("Actualizar carta vinos", 25, 3, "Sala", hoy.plusDays(7));
    agregarTareaInterna("Revisar reservaciones", 20, 1, "Administración", hoy);
    agregarTareaInterna("Capacitar meseros", 90, 3, "Administración", hoy.plusDays(14));
    agregarTareaInterna("Preparar postres", 40, 2, "Cocina", hoy.plusDays(1));
}

private void agregarTareaInterna(String nombre, int tiempo, int prioridad, String departamento, LocalDate fechaEntrega) {
    RecursividadDivideVenceras.Tarea tarea = new RecursividadDivideVenceras.Tarea(nombre, tiempo, prioridad, departamento, fechaEntrega);
    tareasMap.put(nextId++, tarea);
}

public boolean agregar(RecursividadDivideVenceras.Tarea tarea) {
    tareasMap.put(nextId++, tarea);
    return true;
}

public boolean eliminarPorId(int id) {
    return tareasMap.remove(id) != null;
}

public List<RecursividadDivideVenceras.Tarea> listar() {
    return new ArrayList<>(tareasMap.values());
}
public List<RecursividadDivideVenceras.Tarea> ordenarPorPrioridadAltoABajo() {
    List<RecursividadDivideVenceras.Tarea> lista = listar();
    lista.sort((t1, t2) -> Integer.compare(t1.prioridad, t2.prioridad));
    return lista;
}

public List<RecursividadDivideVenceras.Tarea> ordenarPorFechaMasProxima() {
    List<RecursividadDivideVenceras.Tarea> lista = listar();
    lista.sort((t1, t2) -> {
        if (t1.fechaEntrega == null && t2.fechaEntrega == null) return 0;
        if (t1.fechaEntrega == null) return 1; // null dates go last
        if (t2.fechaEntrega == null) return -1; // null dates go last
        return t1.fechaEntrega.compareTo(t2.fechaEntrega);
    });
```

Métodos de ordenamiento y búsqueda

Los métodos pueden ordenar de la siguiente manera con las siguientes funciones:

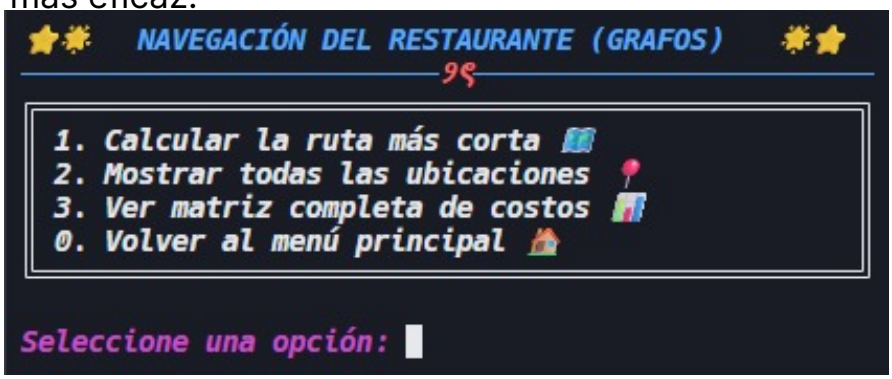
- Con el método de “ordenarPorPrioridadAltoABajo()”
- Se da prioridad ascendente (1 al 3), y desempate por fecha de entrega más próxima.
- Se implementa con el método: Comparator.comparingInt(Tarea::getPrioridad).thenComparing(Tarea::getFechaEntrega, ...).
- Puede ordernar por la fecha más próxima con este método: ordenarPorFechaMasProxima()
- Da prioridad a las fechas de entrega ascendentes (próximas primero).
- Permite usar el metodo buscarPorId(int id):
- Para usar tareasPorId.get(id) dentro del HashMap en búsqueda de orden 1 constante amortizado.
- Llama al metodo buscarPorNombre(String nombre)
- Crea una vista ordenada por nombre (case-insensitive) y aplica búsqueda binaria.
- Retorna todas las coincidencias similares (mismos nombres).

```
// Method to manage tasks using HashMap
private static void gestionarTareas() {
    int op;
    final DateTimeFormatter FMT = DateTimeFormatter.ISO_LOCAL_DATE; // Define formatter here

    do {
        System.out.println(Colores.AZUL + "\n⭐🌟    GESTIÓN DE TAREAS (HASHMAP)    🌟⭐" + Colores.RESET);
        System.out.println(Colores.AZUL + "————————————————————" + Colores.ROJO + "୨୧" + Colores.AZUL + "————————————————————" + Colores.RESET);
        System.out.println(Colores.BLANCO + "╔═══════════════════════════════════════════╗" + Colores.RESET);
        System.out.println(Colores.BLANCO + "┃ 1. Agregar Tarea ➕                       ┃");
        System.out.println("┃ 2. Eliminar Tarea por ID 🗑️               ┃");
        System.out.println("┃ 3. Listar todas las Tareas 📋             ┃");
        System.out.println("┃ 4. Ordenar por Prioridad (alta→baja) 📊   ┃");
        System.out.println("┃ 5. Ordenar por Fecha (próxima 1º) 📅      ┃");
        System.out.println("┃ 6. Buscar por ID (O(1) HashMap) 🔍        ┃");
        System.out.println("┃ 7. Buscar por Nombre 🔎                   ┃");
        System.out.println("┃ 0. Volver al menú principal 🏠            ┃" + Colores.RESET);
        System.out.println(Colores.BLANCO + "╚═══════════════════════════════════════════╝" + Colores.RESET);
        System.out.print(Colores.MORADO + "\nSeleccione una opción: " + Colores.RESET + Colores.BLANCO);
```

Grafos

Y para finalizar, el uso de los grafos fue para el diseño y un sistema de navegación para planificar rutas entre mesas, en el cual se pueden ver las rutas más cortas a cada localización dependiendo en donde se encuentre el empleado y a donde quiere llegar, también ayuda a dar una idea de las dimensiones del local, y adiestrar a los nuevos empleados de una manera más eficaz.



También incluye un método para mostrar la matriz completa y ver como está estructurado en general.

```
private static final int INF = 1_000_000_000;
private final Map<String, Integer> idx = new HashMap<>();
private final List<String> nombres = new ArrayList<>();
private int[][] dist;
private int[][] next;
```

Aquí se utilizan los HashMaps como un índice principal del grafo y las listas almacenan los vecinos directos de cada nodo, esta estructura forma la lista de adyacencia y tiene grandes ventajas como una búsqueda bastante rápida de los HashMaps para encontrar cualquier nodo y la eficiencia de memoria y simplicidad de las listas para manejar sus conexiones.

Conclusión

El desarrollo del sistema de gestión para el restaurante "Dolce Alba" ha demostrado de manera fehaciente cómo las estructuras de datos y algoritmos fundamentales de la informática, a menudo estudiados en teoría, son herramientas indispensables para resolver problemas complejos del mundo real. Este proyecto no solo consistió en escribir código, sino en modelar la dinámica de un restaurante, traduciendo sus operaciones en soluciones lógicas y eficientes.

Esto demuestra que no existe una única "mejor" estructura de datos. La verdadera potencia se logra al combinar sinérgicamente múltiples estructuras y algoritmos, eligiendo la herramienta adecuada para cada desafío específico. El grafo modela el espacio físico, los árboles organizan la jerarquía humana, las tablas hash actúan como un índice universal, y los algoritmos de ordenamiento y recursividad optimizan los flujos de trabajo.
En conclusión, se ha construido con éxito un sistema integral que no solo gestiona, sino que optimiza activamente las operaciones de un restaurante, demostrando que la aplicación inteligente de la teoría informática es la clave para desarrollar soluciones de software robustas, eficientes y de alto impacto.

¿Alguna pregunta?
Comunícate con nosotros.

www.dolcealba.com

restaurant@dolcealba.com

+57 321 456 7890